Sample posts from blog.siliconvalve.com

Simon Waight

April 29, 2024

Contents

List of Figures

1 Cross-posting blog posts to Mastodon, Twitter and LinkedIn using Azure Logic Apps

Welcome! If you're reading this, then I have successfully implemented my plan to build a content cross-posting service using Azure Logic Apps!

In the 10 years that I've been blogging I've learnt that the easiest way to get people across your latest content is to proactively publish alerts to various social platforms. Traditionally I've only ever used Twitter, LinkedIn and dev.to but am increasing my publication to Mastodon.

I blog using the SaaS-version of Wordpress from Wordpress.com which at the subscription level I have provides only limited social endpoints to publish alerts to.

A developer loves nothing if not a good challenge, so let's take a look at how I can build a cross-posting solution using Azure Logic Apps!

1.0.1 Mastodon custom connector for Logic Apps

The first thing I'd usually look to do to make my life easier would be to build a custom Logic Apps connector for the Mastodon API, similar to what I did previously with the Meetup API.

At time of writing this post, the Mastodon team are still working on an autogenerated OpenAPI spec for app developers, so I've decided not to use one of the existing static specifications to avoid having to rework a connector in future.

Regardless, we can easily use Logic Apps with Mastodon to post content (also known as a 'toot'!). Let's take a look at how!

1.0.2 Setup a Mastodon application identity

In order to post the content I have I will need to use the API published by Mastodon, specifically the Post new status endpoint.

This API requires authentication and authorisation in order to be used, so I have to setup a client registration for my Logic App which provides it with privleges to post on my behalf.

To set this up you need to visit the Mastodon server your account is homed on and then use the Development option (1) to create a new application (2).

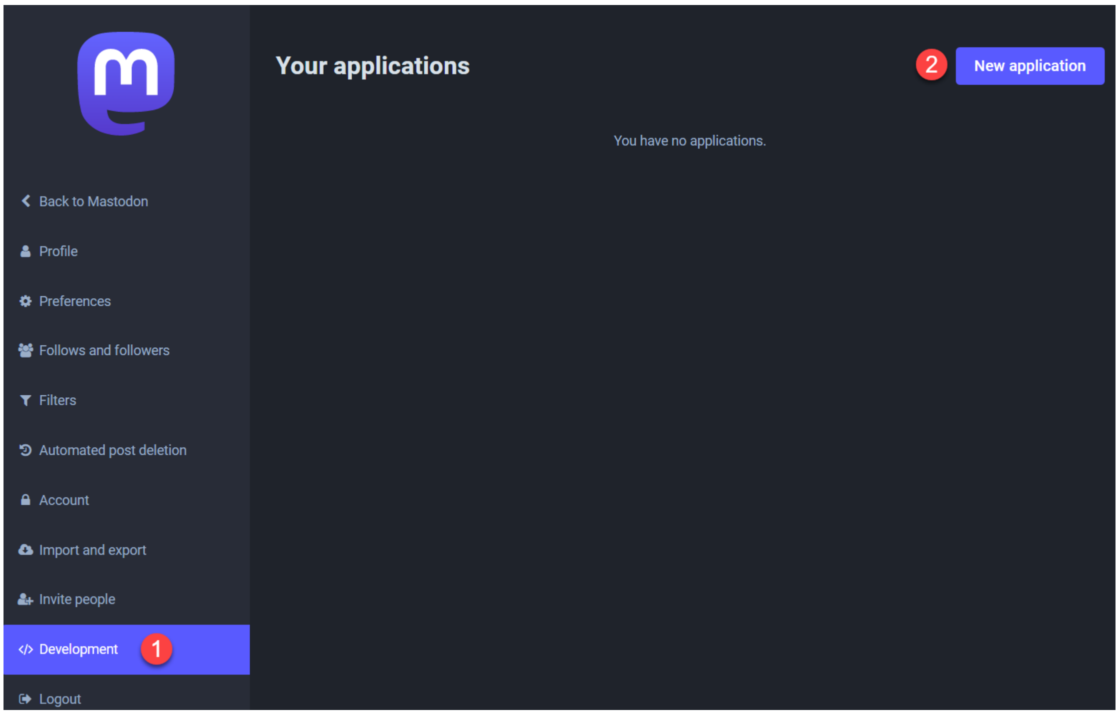
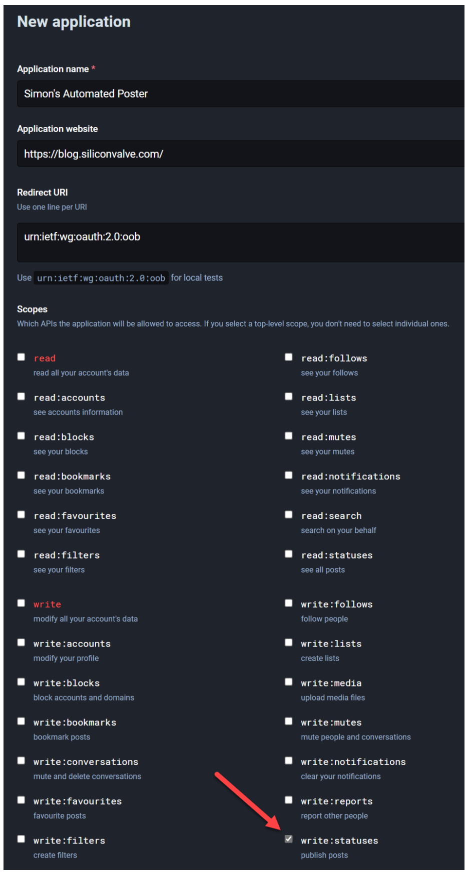


Figure 1: Screenshot of the Mastodon Development menu screen.

Give the application a meaningful name and set the permission scope accordingly. In my case I will only post new items, so I select the `write:statuses` scope (shown by arrow in image) for this application.

Important note: put a real web address in the application website field as this value is used along with the application name in any posts made on Mastodon (see below).



New application

Application name *

Simon's Automated Poster

Application website

https://blog.siliconvalve.com/

Redirect URI

Use one line per URI

urn:ietf:wg:oauth:2.0:oob

Use `urn:ietf:wg:oauth:2.0:oob` for local tests

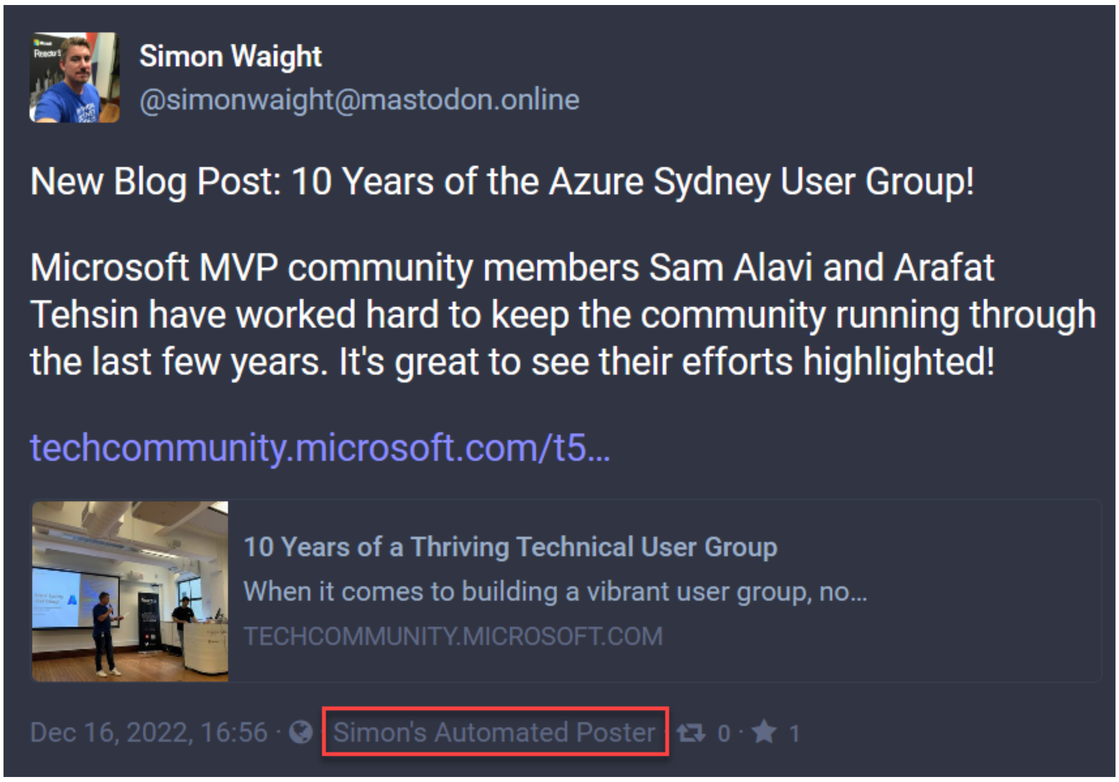
Scopes

Which APIs the application will be allowed to access. If you select a top-level scope, you don't need to select individual ones.

`read` read all your account's data	`read:follows` see your follows
`read:accounts` see accounts information	`read:lists` see your lists
`read:blocks` see your blocks	`read:mutes` see your mutes
`read:bookmarks` see your bookmarks	`read:notifications` see your notifications
`read:favourites` see your favourites	`read:search` search on your behalf
`read:filters` see your filters	`read:statuses` see all posts
`write` modify all your account's data	`write:follows` follow people
`write:accounts` modify your profile	`write:lists` create lists
`write:blocks` block accounts and domains	`write:media` upload media files
`write:bookmarks` bookmark posts	`write:mutes` mute people and conversations
`write:conversations` mute and delete conversations	`write:notifications` clear your notifications
`write:favourites` favourite posts	`write:reports` report other people
`write:filters` create filters	[✓ `write:statuses`] publish posts

Figure 2: Mastodon application registration page with write:statuses scope selected.

Here's a sample post with the application name highlighted. You can click on this and be taken to the web address supplied.



Simon Waight
@simonwaight@mastodon.online

New Blog Post: 10 Years of the Azure Sydney User Group!

Microsoft MVP community members Sam Alavi and Arafat Tehsin have worked hard to keep the community running through the last few years. It's great to see their efforts highlighted!

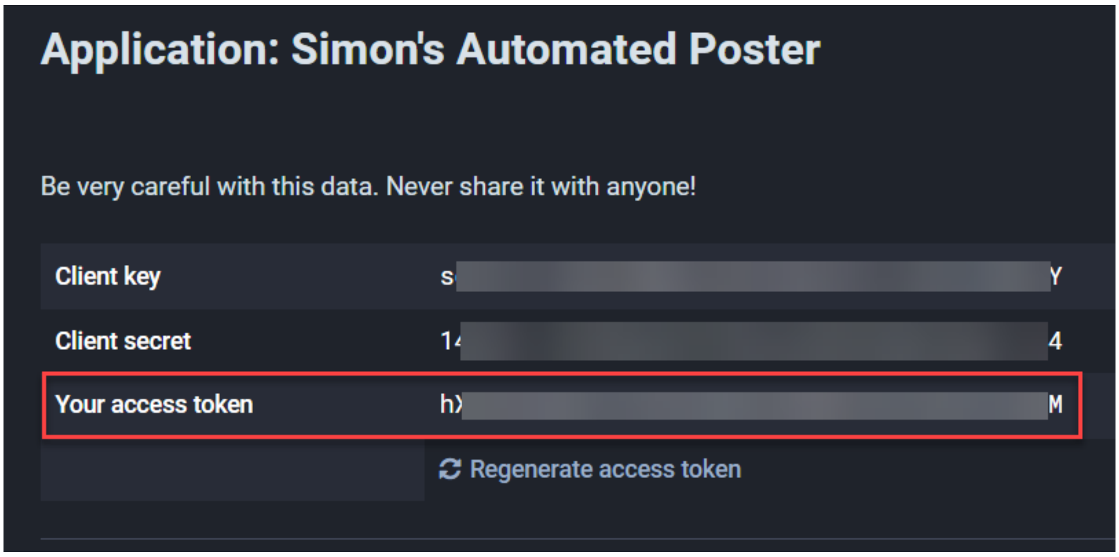
techcommunity.microsoft.com/t5…

10 Years of a Thriving Technical User Group
When it comes to building a vibrant user group, no…
TECHCOMMUNITY.MICROSOFT.COM

Dec 16, 2022, 16:56 · Simon's Automated Poster · 0 · 1

Figure 3: Sample Mastodon post with client application name highlighted.

Once I save the Application the Mastodon server returns the registration details for the application, which will seem familiar if you've used OAuth previously. These are super sensitive, so don't share with anyone (just like it says in the screenshot!)



Application: Simon's Automated Poster

Be very careful with this data. Never share it with anyone!

Client key	s[redacted]Y
Client secret	1[redacted]4
Your access token	h[redacted]M
	Regenerate access token

Figure 4: Application registration displaying client details and keys.

Now I have a token (highlighted in red) that I need to call the Mastodon API so let's progress on to building our Logic App solution.

1.0.3 Logic App Message Distributor

Ideally I'd like to drive my final solution off of a webhook tied to the Post publication event on Wordpress, but this feature isn't available for my Wordpress subscription, so instead I use a simple HTTP trigger in my Logic App.

The benefit of this approach is I allow any upstream service to post to it, which means if I ever move off Wordpress I could fairly easily do something like wire a call to the Logic App into, say, a GitHub Action if I were looking to publish a static website.

The first step I need to take is define what data I want to send out to all the services I publish to. For the purpose of this blog post I am going to use the following simple JSON format when calling my Logic App.

```json
{
  "Title": "A sample blog post title",
  "Summary": "This is a test blog post summary.",
  "ImageRef": "https://link.to/image/for/share",
  "Link": "https://blog.siliconvalve.com/"
}
```

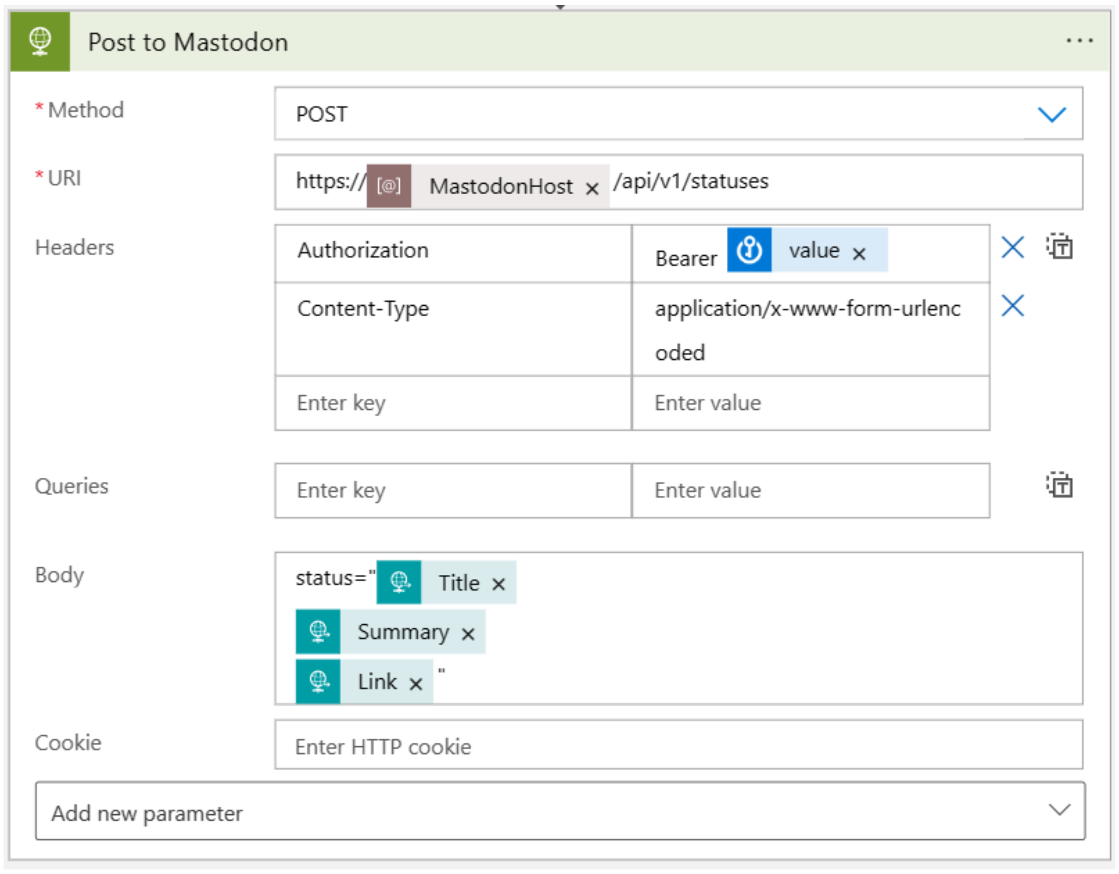
The great thing in using JSON is that the HTTP trigger in the Logic App will automatically parse the JSON and turn each property into a variable in my Logic App that I can easily reference in Actions.

So the solution is reusable I am going to create a Parameter for the Logic App - MastodonHost - which will hold the hostname on the Mastodon server we want to post to ('mastodon.online' in my case).

Additionally, as the Mastodon token is sensitive, even with a restricted scope, I am still going to place it into an Azure Key Vault secret that can be read by the Logic App at runtime using a managed service identity.

1.0.4 Posting to Mastodon

Now we have all the bits in place we can go ahead and use the standard HTTP action to invoke the Mastodon API.



Post to Mastodon

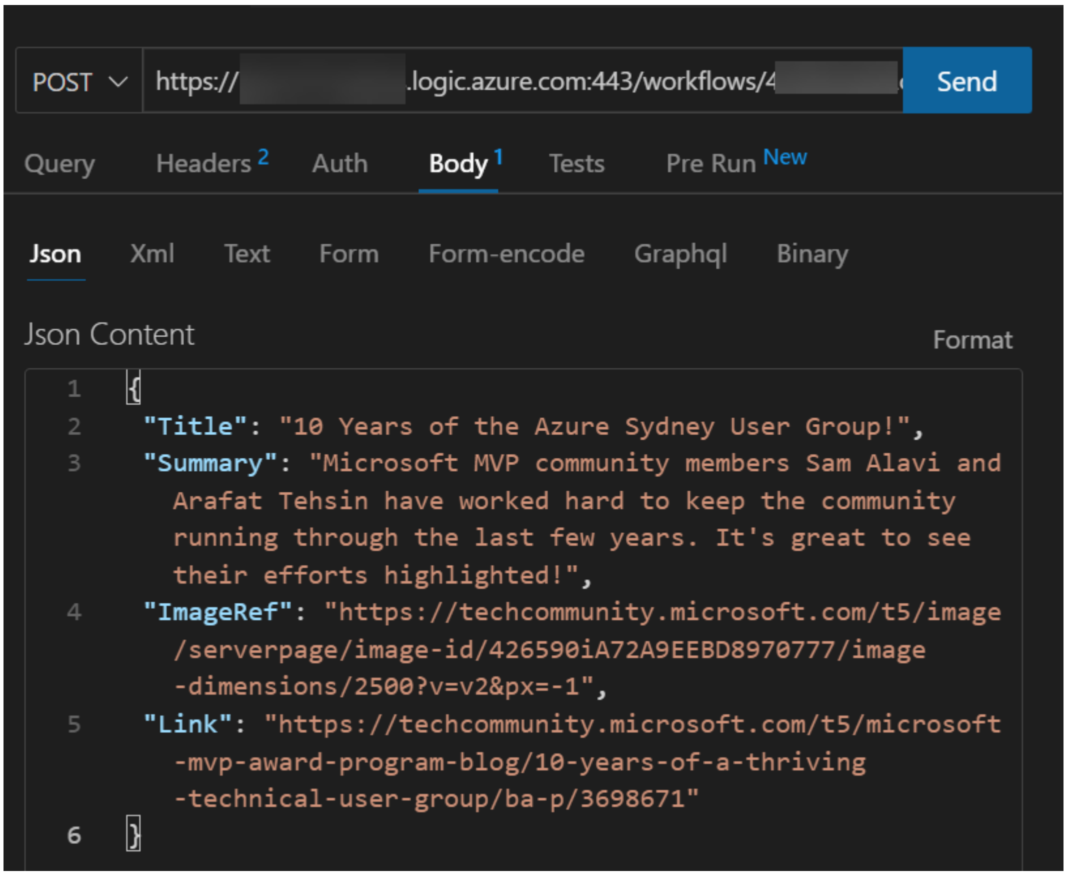
* Method	POST	
* URI	https://[@] MastodonHost /api/v1/statuses	
Headers	Authorization	Bearer value
	Content-Type	application/x-www-form-urlencoded
	Enter key	Enter value
Queries	Enter key	Enter value
Body	status=" Title Summary Link "	
Cookie	Enter HTTP cookie	

Add new parameter

Figure 5: Logic App HTTP Action configured to post a message on Mastodon.

The key vault secret value is passed in as part of the HTTP Authorization header as an OAuth Bearer token and we set the content type appropriately. The body of the message contains the status we wish to post. There are lots of other types we can publish (media and polls), but for my use case I simply want to publish some text and a link.

If I want to post to Mastodon I can use a simple REST client such as Thunderclient to test it out. You can see the resulting post above.



https://[redacted].logic.azure.com:443/workflows/4[redacted]

```
{
  "Title": "10 Years of the Azure Sydney User Group!",
  "Summary": "Microsoft MVP community members Sam Alavi and
    Arafat Tehsin have worked hard to keep the community
    running through the last few years. It's great to see
    their efforts highlighted!",
  "ImageRef": "https://techcommunity.microsoft.com/t5/image
    /serverpage/image-id/426590iA72A9EEBD8970777/image
    -dimensions/2500?v=v2&px=-1",
  "Link": "https://techcommunity.microsoft.com/t5/microsoft
    -mvp-award-program-blog/10-years-of-a-thriving
    -technical-user-group/ba-p/3698671"
}
```

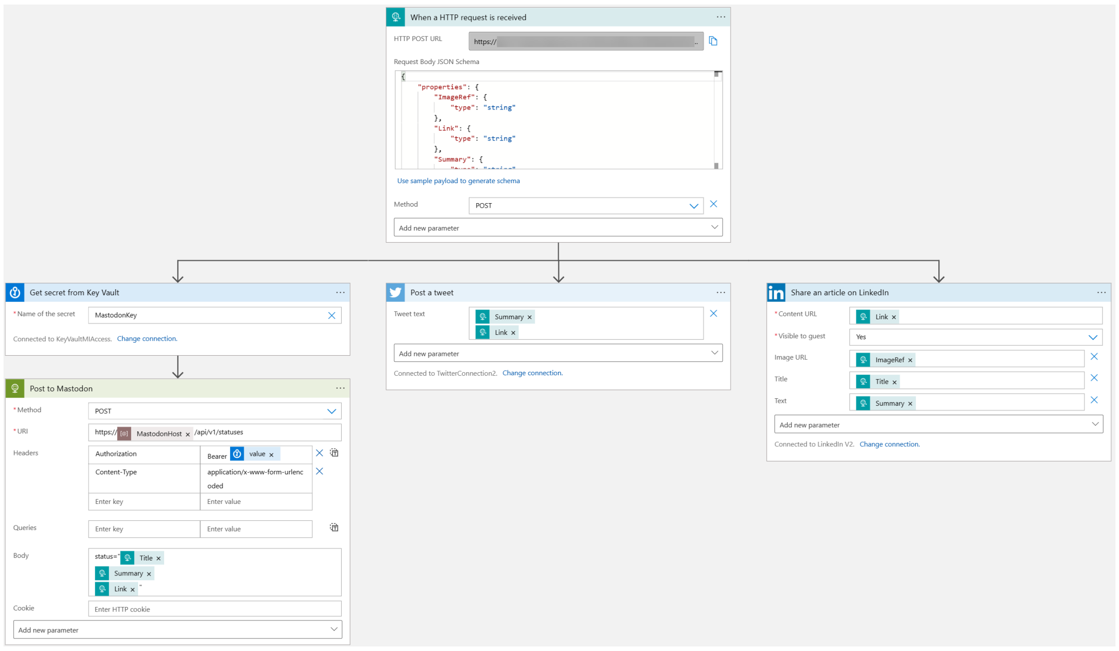
Figure 6: Simple REST request to pust a post into Mastodon.

1.0.5 Extending the idea

Developers love the concept of Don't Repeat Yourself (DRY), so let's extend our posting Logic App to help us in other places. For my example I've added Twitter and LinkedIn cross-posting which is done in parallel to my Mastodon post.

I'm not going to go into depth how to add these steps as they are well documented elsewhere (Twitter and LinkedIn).

The full solution is visualised below - click to enlarge.



When a HTTP request is received

HTTP POST URL: https://[redacted]..

Request Body JSON Schema

```
{
    "properties": {
        "ImageRef": {
            "type": "string"
        },
        "Link": {
            "type": "string"
        },
        "Summary": {
```

Use sample payload to generate schema

Method: POST

Add new parameter

Get secret from Key Vault

* Name of the secret: MastodonKey

Connected to KeyVaultMIAccess. Change connection.

Post to Mastodon

* Method: POST

* URI: https:// MastodonHost /api/v1/statuses

Headers	
Authorization	Bearer value
Content-Type	application/x-www-form-urlencoded
Enter key	Enter value

Queries: Enter key | Enter value

Body: status=" Title Summary Link "

Cookie: Enter HTTP cookie

Add new parameter

Post a tweet

Tweet text: Summary Link

Add new parameter

Connected to TwitterConnection2. Change connection.

Share an article on LinkedIn

* Content URL: Link

* Visible to guest: Yes

Image URL: ImageRef

Title: Title

Text: Summary

Add new parameter

Connected to LinkedIn V2. Change connection.

Figure 7: Azure Logic App that cross posts content on Mastodon, Twitter and LinkedIn.

1.0.6 Wrapping it up

So, there we are - a quick no-code way to take a single piece of text, hyperlink (and an optional image) and post it to Mastodon, Twitter and LinkedIn. You can grab the bicep deployment template for this solution from the GitHub repository it is stored in.

The beauty of spinning this solution up in a Consumption Plan for Logic Apps is that I will only pay when I invoke the API which means there is minimal cost for the value it adds.

Now I just need to disable the social auto-posting from Wordpress!

Happy Days!

2 Fix Logic App Connections Managed Identity errors in Bicep templates

Off the back of my recent post on building an Azure Logic Apps solution for cross-posting on social media I wanted to export what I'd quickly built and make an easily deployable asset in Azure using Bicep. Right now there is no way to export deployed artefacts from Azure directly to Bicep - you must first export to an Azure Resource Manager (ARM) template and then use the Bicep tooling to convert.

The majority of the time the result is a Bicep template that requires a little tidy up before it can be used, however, once I'd tidied up my solution, I went to test the Bicep template and immediately hit upon this error:

> "The API connection 'keyvault' is not configured to support managed identity."

Replace **keyvault** with whatever the name of the Logic App API Connection is you are using in Azure.

2.0.1 Create a User Assigned Managed Identity

First off, I thought this was due to my use of a System-Assigned Managed Identity which is created and managed transparently by an Azure resource such as a Logic App. The beauty of this type of Managed Identity is that if you delete the resource it was created by, the Managed Identity is also deleted.

To fix this I thought I'd add a User-Assigned Managed Identity to the Bicep file, so I went ahead and added the below.

```
resource managed_service_identity_kv_resource 'Microsoft.ManagedIdentity/userAssignedIdentities@2
  name: managed_identity_name
  location: resource_group_location
}
```

Now I have a Managed Identity that I can reference elsewhere in the Bicep file without creating a circular dependency.

2.0.2 Allow Managed Identity to access Key Vault

Next up I can reference this Managed Identity when I create my Key Vault instance. See line 13 in the below GitHub Gist.

```
resource key_vault_resource 'Microsoft.KeyVault/vaults@2022-07-01' = {
  name: key_vault_name
  location: resource_group_location
  properties: {
    sku: {
      family: 'A'
      name: 'standard'
    }
    tenantId: subscription().tenantId
    accessPolicies: [
      {
        tenantId: subscription().tenantId
        objectId: managed_service_identity_kv_resource.properties.principalId
        permissions: {
          certificates: []
          keys: []
          secrets: [
            'get'
          ]
        }
      }
    ]
    enabledForDeployment: false
    enabledForDiskEncryption: false
    enabledForTemplateDeployment: false
    enableSoftDelete: true
    softDeleteRetentionInDays: 7
    enableRbacAuthorization: false
    publicNetworkAccess: 'Enabled'
  }
}
```

2.0.3 Logic App Parameters

Our connections will be listed in the Parameters section of our Logic App in the Bicep template, and we should find that the authentication type is setup to **ManagedServiceIdentity** as per line 9 below.

```
parameters: {
      '$connections': {
        value: {
          keyvault: {
            connectionId: connections_keyvault_resource.id
            connectionName: 'keyvault'
            connectionProperties: {
              authentication: {
                type: 'ManagedServiceIdentity'
              }
            }
            id: subscriptionResourceId('Microsoft.Web/locations/managedApis', resource_group_loca
          }
        }
      }
 }
```

Sadly, none of this solved my issue.

2.0.4 Connection definition

This is where the magic happens. There is currently a gap in ARM and Bicep's ability to export / define the values required to correctly map a Connection to a Managed Identity (there's an open bug on GitHub on it). Here's our Connection definition in the Bicep file.

```
resource connections_keyvault_resource 'Microsoft.Web/connections@2016-06-01' = {
  name: connections_keyvault_name
  location: resource_group_location
  properties: {
    displayName: 'KeyVaultMIAccess'
    parameterValueType: 'Alternative'
    alternativeParameterValues: {
      vaultName: key_vault_name
    }
    customParameterValues: {}
    api: {
      name: 'keyvault'
      displayName: 'Azure Key Vault'
      description: 'Azure Key Vault is a service to securely store and access secrets.'
      iconUri: 'https://connectoricons-prod.azureedge.net/releases/v1.0.1597/1.0.1597.3005/keyvau
      brandColor: '#0079d6'
      id: subscriptionResourceId('Microsoft.Web/locations/managedApis', resource_group_location,
      type: 'Microsoft.Web/locations/managedApis'
    }
  }
}
```

The key lines that solve our issue are as follows. This entry is required to map the Connection to the Key Vault.

`parameterValueType: 'Alternative' alternativeParameterValues: { vaultName: key_vault_name }`

So, there we go. We can now you can deploy Azure Logic Apps that use a Managed Service Identity to connect to an Azure Key Vault!

I'm chalking this post up as "would have saved me a bunch of time" because it took a look of looking around to get to the ultimate answer. I hope this one saves you some time! Find the full Bicep file on GitHub.