**Using PowerShell with REST APIs**

With the advent of cloud services, PowerShell is more important than ever. The Invoke-RestMethod and Invoke-WebRequest functions of PowerShell allow easy interactions with an API and aid in creating complex workflows. There are many different authentication methods and API call structures that exist. The Invoke-RestMethod PowerShell function can easily handle any number of different scenarios.

**Invoke-RestMethod vs Invoke-WebRequest**

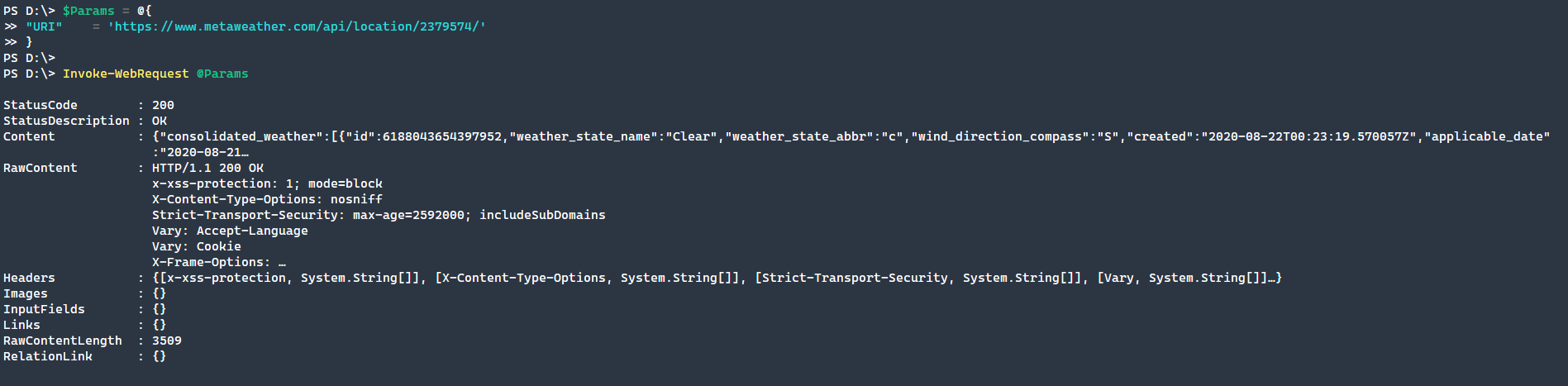
The two most common PowerShell methods of interacting with REST API’s are to use either Invoke-RestMethod or Invoke-WebRequest. To interact with a REST API the Invoke-RestMethod function is a clear choice. This function natively turns input JSON or XML into native PowerShell objects that make further interaction easy. Invoke-WebRequest is best for raw interactions with websites. This will return the output structure of a website and the raw JSON of an API which necessitates further transformations.

Here we are going to show the differences in output. Both cmdlets will be used to query an unauthenticated weather API to see how the output is shown in PowerShell.

**Invoke-WebRequest**

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5 | $Params = @{  "URI"    = '<https://www.metaweather.com/api/location/2379574/>'  }    Invoke-WebRequest @Params |

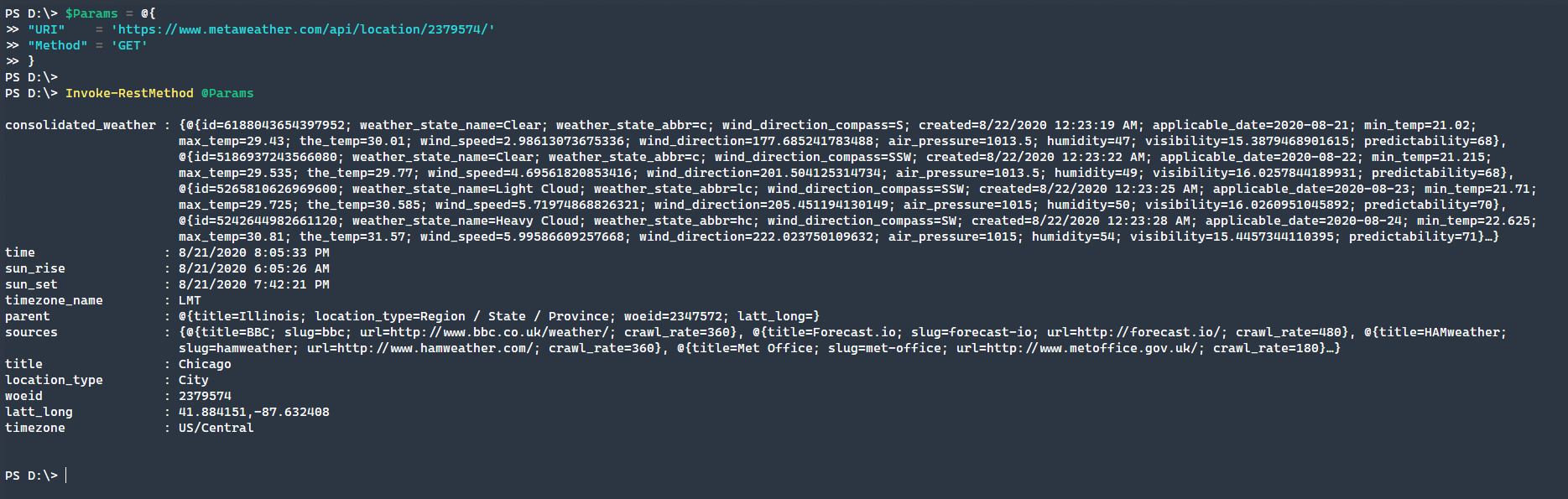


The results from the Invoke-WebRequest call includes header, status, and raw contents. To convert the content to a proper JSON object you will need to call ConvertTo-JSON. Now let’s take a look at how Invoke-RestMethod handles the same API call.

**Invoke-RestMethod**

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $Params = @{  "URI"    = '<https://www.metaweather.com/api/location/2379574/>'  "Method" = 'GET'  }    Invoke-RestMethod @Params |



As you can see in the response, the JSON output is automatically converted to a PowerShell object. For REST API’s this conversion greatly simplifies the management and use.

**API Authentications**

Of course, with most API calls at some point authentication will be needed. There are several different methods that work for an API each with their advantages and disadvantages.

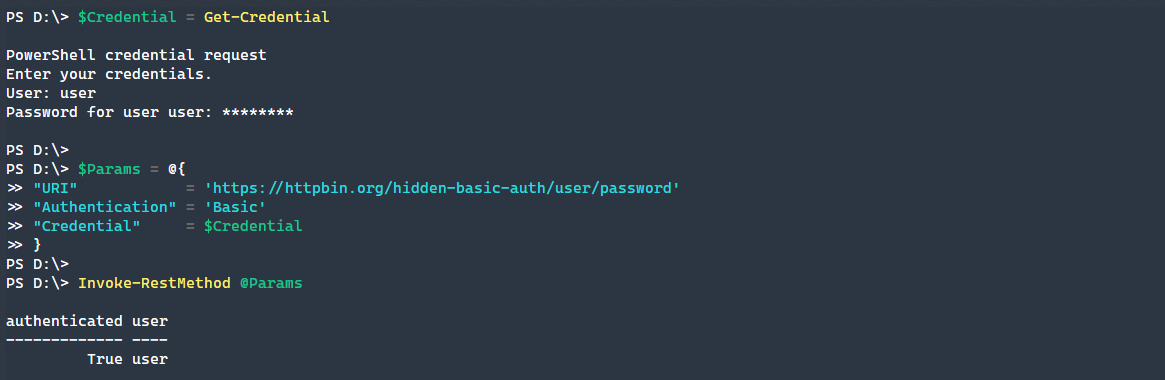
* **Basic** – Every API call has the username and password sent with the call.
* **API Key** – An API is usually sent from a specific header entry, or sometimes within the body of the request.
* **OAuth** – This uses a user-initiated token, but is difficult to implement.

**Basic Authentication**

Used in the early days of web authentication, passing user credentials is not the most common method as there are more secure methods available. If the credentials were read during transit, the account would be easily compromised and not quickly revoked. User credentials are not as commonly rotated or locked down as an API key is.

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $Credential = Get-Credential    $Params = @{  "URI"            = '<https://httpbin.org/hidden-basic-auth/user/password>'  "Authentication" = 'Basic'  "Credential"     = $Credential  }    Invoke-RestMethod @Params |

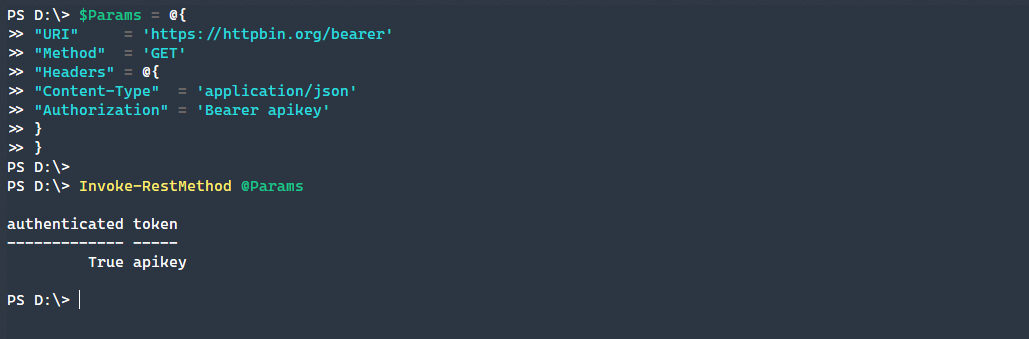


**API Key Authentication**

As basic authentication is generally known to be insecure, the next best method is using a standard Authorization API key header. Using this method means that you typically assign API keys to a user with specific permissions, rotate keys more readily, and control expiration dates better.

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | $Params = @{  "URI"     = '<https://httpbin.org/bearer>'  "Method"  = 'GET'  "Headers" = @{  "Content-Type"  = 'application/json'  "Authorization" = 'Bearer apikey'  }  }    Invoke-RestMethod @Params |



An alternative to this header format is a custom header value that is often specifically tied to the REST API. In this example below, we use X-API-Authentication to send the API key.

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5 | @{  "Content-Type"  = 'application/json'  "X-API-Authentication" = 'customapikey'  } |

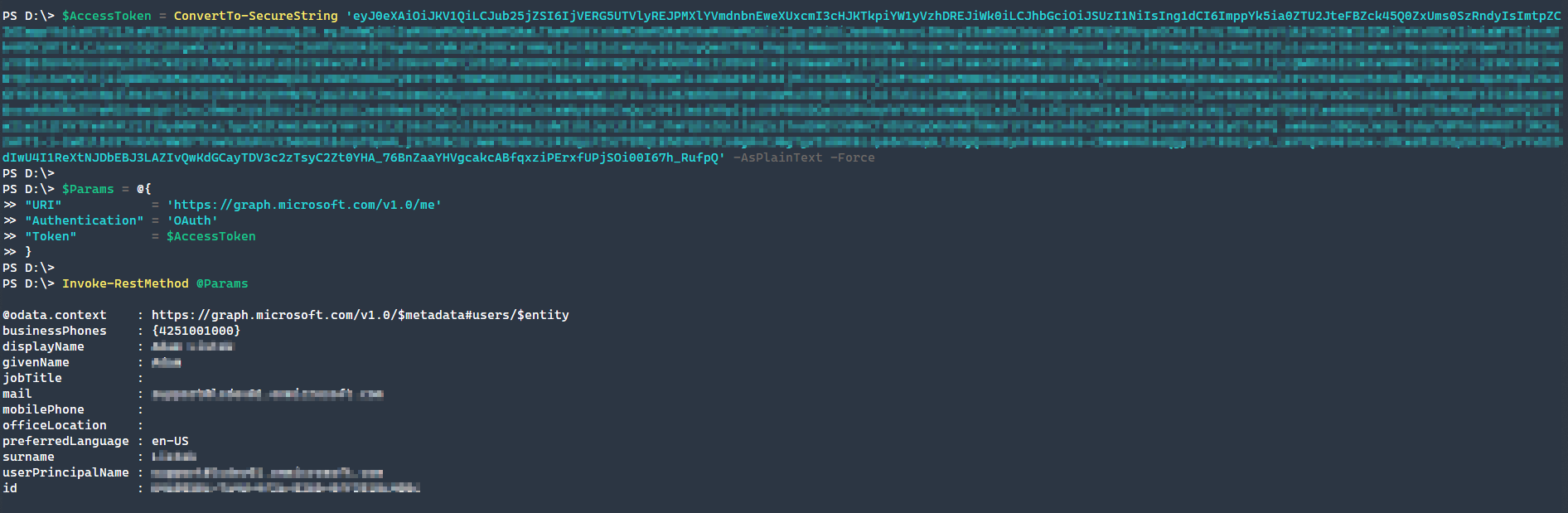
**OAuth Authentication**

To demonstrate how Invoke-RestMethod works with OAuth we are going to demonstrate using the Microsoft Graph Explorer. Navigate to the following test URL below:

<https://developer.microsoft.com/en-us/graph/graph-explorer/preview>

Click on **Sign in to Graph Explorer** and click on the **Access Token** tab to retrieve the token. As you can see from the results shown below, utilizing the access token returns the user identity from the Microsoft Graph API.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $AccessToken = ConvertTo-SecureString 'eyJ0eXAiOiJKV1QiL...' -AsPlainText -Force    $Params = @{  "URI"            = '<https://graph.microsoft.com/v1.0/me>'  "Authentication" = 'OAuth'  "Token"          = $AccessToken  }    Invoke-RestMethod @Params |



**Interacting with an API**

To demonstrate a more practical example of an end-to-end script that interacts with the Microsoft Graph API, let’s retrieve a group’s members. Unlike the above demonstration using the Graph API Explorer, we are going to demonstrate this process using a more production-ready method.

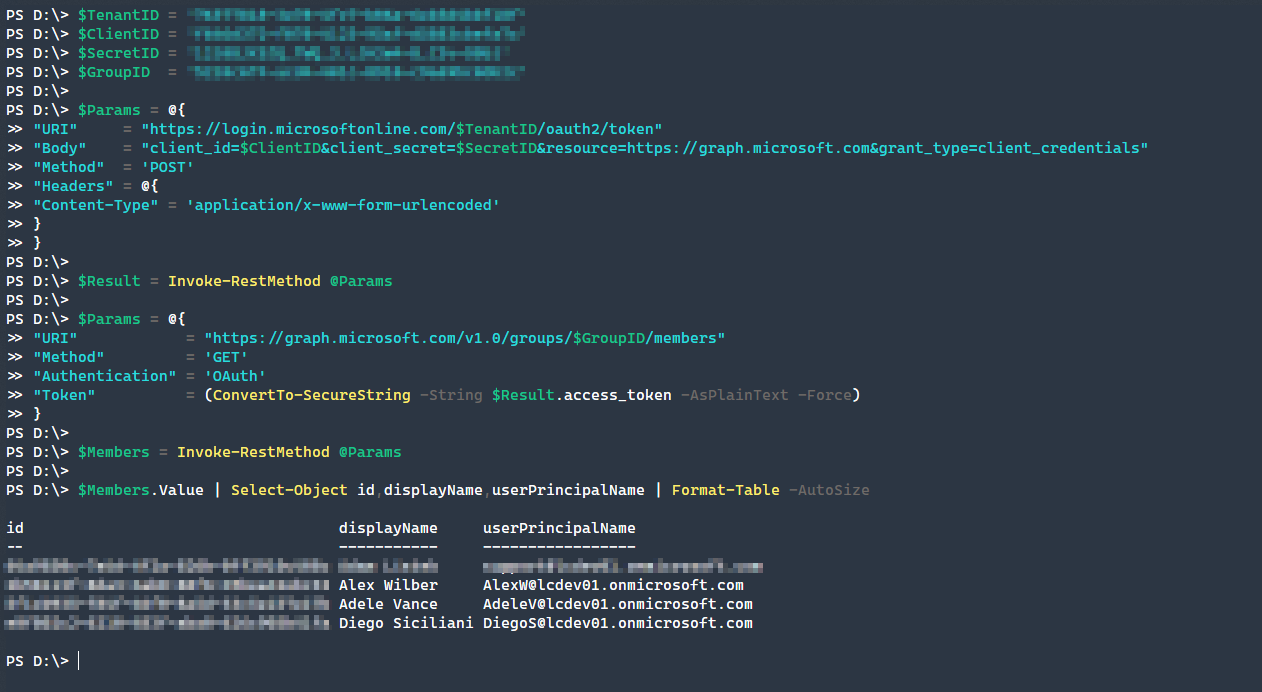
Though we are not going into the full details on how to do this, in this article, you will need to have created an App Registration in Azure AD and saved the Client ID, Secret ID, and Tenant ID to properly authenticate. The script does the following steps to retrieve a groups members.

* Retrieve the time-limited access token
* Using the returned access token request the group members
* Format the returned PowerShell object

The process outlined below is typical of how many API’s are utilized and interacted with. With the native PowerShell object return of Invoke-RestMethod it is easy to further manipulate the values from a given endpoint.

PowerShell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | # Values shown below are randomly generated, but will need to reflect your environment  $TenantID = 'f5181690-85e8-454e-9955-ca00577a1725'  $ClientID = 'eb48df6b-aec4-401a-bdb6-7977cbadebc7'  $SecretID = 'yn1YFeRvDd2wTul6ZLnmAoQBkjRpuTb'  $GroupID  = '6b85f3b5-a879-41fa-a36f-497db70f7660'    $Params = @{  "URI"     = "<https://login.microsoftonline.com/$TenantID/oauth2/token>"  "Body"    = "client\_id=$ClientID&client\_secret=$SecretID&resource=https://graph.microsoft.com&grant\_type=client\_credentials"  "Method"  = 'POST'  "Headers" = @{  "Content-Type" = 'application/x-www-form-urlencoded'  }  }    $Result = Invoke-RestMethod @Params    $Params = @{  "URI"            = "<https://graph.microsoft.com/v1.0/groups/$GroupID/members>"  "Method"         = 'GET'  "Authentication" = 'OAuth'  "Token"          = (ConvertTo-SecureString -String $Result.access\_token -AsPlainText -Force)  }    $Members = Invoke-RestMethod @Params    $Members.Value | Select-Object id,displayName,userPrincipalName | Format-Table -AutoSize |



As you can see from the above results, we have four group members returned from the Microsoft Graph API. At this point, you can further use their ID values to call on a user endpoint, for example, to gain further information or perform an action, such as removing a user from a group. Any subsequent API calls will use the same access token for as long as it remains valid.

**Conclusion**

With Invoke-RestMethod you can address nearly every scenario that may come up when interacting with an API. Many cloud services tie together different components of an ecosystem and PowerShell works well to tie them all together!