Graphing M365: Lab Four

# Objective

In this lab, we automate graphing Microsoft 365 objects with PowerShell.

# Procedure

## Setting up the code environment

1. Create a new code directory somewhere on your system. This should be within your user’s control on the machine and memorable.
2. Within the directory, create three new folders:
   1. Classes
   2. Functions
   3. Output
3. Load your preferred code editor. If you are relatively new to PowerShell, the built in PowerShell ISE (included on all Windows Systems) is a great start. VS Code, which is cross platform, is also decent and has full PowerShell syntax support.
4. Regardless of your code editor, I do recommend running PowerShell on its own apart from the editor to help limit troubleshooting issues later.

## Creating our base classes

1. Within the Classes directory, create and open a file named Nodes.psm1. PSM1 files are PowerShell module files which are intended to be loaded by scripts that end in PS1.
2. Copy the code below to create the base class for all nodes.

*class* BasicNode {

    [*string*]$Type # Represents the type of node

    [*string*]$Id # This should map to the node id from MSFT

    [*string*]$DisplayName # This should map to the DisplayName from MSFT

    BasicNode(){} # This allows instantiation of the node

    BasicNode( # This constructor takes in three args

        [*string*]$Type*,*

        [*string*]$Id*,*

        [*string*]$DisplayName #

    ) {

        $this.Type *=* $Type

        $this.Id *=* $Id

        $this.DisplayName *=* $DisplayName

    }

}

1. Save the file.
2. In the classes directory, create and open a file named Edges.psm1.
3. Copy the code below to create the base class for all edges

*class* BasicEdge {

    [*string*]$Relationship *# Identifies the type of relationship*

    [*string*]$Source       *# The node which "owns" the relationship*

    [*string*]$Target       *# The node which "receives" the relationship*

    BasicEdge(){}

    BasicEdge(

        [*string*]$Relationship*,*

        [*string*]$Source*,*

        [*string*]$Target

    ) {

        $this.Relationship *=* $Relationship

        $this.Source *=* $Source

        $this.Target *=* $Target

    }

}

1. Save the Edges.psm1 file. This file will not be used again for this course.

## Creating function files

1. Navigate to the Functions folder created in the first section of the lab.
2. Create and Open the file **NodeFunctions.psm1**. This will contain all functions related to fetching and modifying nodes.
3. Import the Node definitions created in section two with the code below.

Using Module ..\Classes\Nodes.psm1

*Note: the standard “Import-Module” cmdlet does not import classes. Therefor, we must use the Using Module <module> to get the classes created in the Nodes.psm1 module. Using statements must be declared before any other code in a PowerShell script unlike modules which may be referenced at any time.*

1. Save the file.
2. Create and **open** the file **EdgeFunctions.psm1**. This will contain all functions related to fetching and modifying edges.
3. Import the Edge definitions created in section two with the code below.

Using Module ..\Classes\Edges.psm1

1. Save the file.
2. Create and Open the file **CypherFunctions.psm1**. This file will contain all functions related to the conversion and export of Cypher queries.
3. Save the file as empty for now.

## Creating the Main Script

1. In the root directory of the project (should contain the directories Classes and Functions), create and open a new file **m365-to-cypher.ps1**. This is the only file with this **.ps1** file ending that we will create during this course.
2. Copy the code below to import all the function and class modules.

*## -- Imports*

Using Module .\Classes\Nodes.psm1

Using Module .\Classes\Edges.psm1

Import-Module .\Functions\NodeFunctions.psm1

Import-Module .\Functions\EdgeFunctions.psm1

Import-Module .\Functions\CypherOutput.psm1

1. Next, below the imports, add in the following variables which will serve to keep memory of all the nodes and edges gathered later.

*## -- Variables*

$ServicePrincipalNodes *=* *@*()

$UserNodes *=* *@*()

$GroupNodes *=* *@*()

$RoleNodes *=* *@*()

$DeviceNodes *=* *@*()

$NamedLocationNodes *=* *@*()

$ConditionalAccessPolicyNodes *=* *@*()

$RawConditionalAccessPolicies *=* *@*()

1. After the variable declarations, add some logic to add some flow to the application. Aesthetics are important to the UX, but the degree to which you take it in this class is up to you.

*## -- Logic*

Clear-Host *# Clears the terminal so all previous info removed*

Write-Host " Starting CA to Cyper.`r`n " *-*ForegroundColor White

Write-Host " Checking prerequisites." *-*ForegroundColor White

1. Next, add in a try catch block to control errors throughout the application run.

*try* {

*## [l] -- Precheck*

    Write-Host " Checking for Microsoft Graph Powershell." *-*ForegroundColor Yellow

*try* {

*If* (*!*(Get-Module *-*Name Microsoft.Graph *-*ListAvailable)) {

            Install-Module *-*Name Microsoft.Graph

        }

    } *catch* {

*throw* " Could not install Microsoft Graph."

    }

    Write-Host "`r`n Connecting to Microsoft Graph." *-*Foreground White

*## [l] -- Connect to Microsoft Graph*

*try* {

        Connect-MgGraph *-*Scopes "Directory.Read.All,User.Read.All,Policy.Read.All,Group.Read.All" *|* Out-Null

        Select-MgProfile *-*Name Beta

        Write-Host " Connected to Microsoft Graph." *-*ForegroundColor Yellow

    } *catch* {

*throw* " Could not connect to Microsoft Graph."

    }

} *catch* { Write-Host $\_ *-*ForegroundColor Red }

1. Finally, after the try/catch block, add some clean up. By removing the modules explicitly at the end of the script, we prevent modules from staying in memory between changes of the script.

*# [l] -- Close the application*

Disconnect-MgGraph *|* Out-Null

Remove-Module *-*Name Nodes

Remove-Module *-*Name Edges

Remove-Module *-*Name NodeFunctions

Remove-Module *-*Name EdgeFunctions

Remove-Module *-*Name CypherOutput

Write-Host "`r`n Quitting CA to Cypher." *-*ForegroundColor Green

1. Save the file and run the application in your PowerShell terminal of choice. From the parent directory, you can run the file by simply typing .\m365-to-cypher.ps1

Text

Description automatically generated

1. When the scripts runs, it will attempt to install the graph module (if needed) and then sign you into Microsoft Graph. Use the credentials created in the second lab with the developer sandbox. Your sign-on email will end with **@\*.onmicrosoft.com**.If you encounter any issues with the code at this point, troubleshoot appropriately.

## Gathering Nodes

1. Open the Nodes.psm1 file in your code editor.
2. Open the spreadsheet from lab two and grab the information about user nodes.

Graphical user interface, text, application

Description automatically generated

1. Right away we can see that the base node type we created covers the first three columns:
   1. Type = Node Type
   2. Id = Unique ID
   3. DisplayName = Human Readable Name
2. Now, use the code below to create a new class that extends the base class.

*class* UserNode : BasicNode {

    UserNode(){}

}

*Note: In the first line, we tell PowerShell we are extending the Basic Node, and so all Basic Node properties are assumed. We then create a line to allow a generic creation of a UserNode without requiring any arguments.*

1. Since the BasicNode class covers all type, id, and name of each node, in our child classes, we only need to add the properties we identified in Lab Two such and listed as “Additional Properties.” In this case:
   1. AccountEnabled
   2. CreatedDateTime
   3. LastPasswordChangeDateTime
   4. Mail
2. As a good rule of thumb, each of these attributes should be typed. For simplicity in this course, all values will be either **bool** or **string**. Given these limits, we can add the additional properties to our new use class.

    [*string*]$AccountCreated

    [*bool*]$AccountEnabled

    [*string*]$LastPasswordChange

    [*string*]$Mail

1. Finally, we need to create a constructor that allows us to pass in all the values at once.

    UserNode(

        [*string*]$Type*,*

        [*string*]$Id*,*

        [*string*]$DisplayName*,*

        [*string*]$AccountCreated*,*

        [*bool*]$AccountEnabled*,*

        [*string*]$LastPasswordChange*,*

        [*string*]$Mail

         ) : base($Type*,* $Id*,* $DisplayName) {

        $this.AccountCreated *=* $AccountCreated

        $this.AccountEnabled *=* $AccountEnabled

        $this.LastPasswordChange *=* $LastPasswordChange

        $This.Mail *=* $Mail

    }

1. With these changes complete. Save the Nodes.psm1 file. The completed file is pictured below.  
   
2. Now open the NodeFunctions.psm1 file.
3. The first thing to create, is a new function for gathering Users from M365 and converting them into our new UserNode class. Let’s start by laying out a generic function that takes no arguments and returns a list of objects (**$Return**).

*Function* Get-UserNodes() {

*try* {

        $Return *=* *@*()

*return* $Return

    }

*catch* {

*throw* $\_

    }

}

1. Within the try block, we need to gather all the users from Microsoft graph. This can be completed with the following code.

$Users *=* Get-MgUser

1. Next, we need to take all the users returned by Microsoft and iterate through them. To provide a better UX, let’s also include a progress bar so we aren’t waiting without feedback.

$Count *=* 0

*ForEach*($User *in* $Users) {

    $Count *+=* 1

    Write-Progress *`*

*-*Id 0 *`*

*-*Activity "Gathering Users" *`*

*-*Status $User.DisplayName *`*

*-*PercentComplete ($Count*\**100 */* $Users.Count)

}

1. Now, this isn’t especially useful as it would only show the display name of each user without returning any values. Let’s fix that by converting each user from Microsoft Graph into a UserNode as defined by the Nodes.psm1 file.

    $Node *=* [*UserNode*]::new(

        "User"*,*

        $User.Id*,*

        $User.DisplayName*,*

        $User.createdDateTime*,*

        $User.AccountEnabled*,*

        $User.LastPasswordChangeDateTime*,*

        $User.Mail

    )

    $Return *+=* $Node

1. Finally, before the return statement, we need to terminate the progress bar so it doesn’t persist until the script finishes.

        Write-Progress *`*

*-*Id 0 *`*

*-*Activity "Gathering Users" *`*

*-*Status "Complete" *`*

*-*Completed

1. This completes the changes needed for the NodeFunctions.psm1 file. It should look like the image below.

Text

Description automatically generated

1. Save the NodeFunctions.psm1 file and open the **m365-to-cypher.ps1** script file.
2. Start by creating a new code block for gathering user information. Insert this code block at the end of the try block we created before.

*## [l] -- Start enumerating user accounts*

*try* {

        Write-Host " Enumerating Users." *-*ForegroundColor Yellow

    } *catch* {

        Write-Host $\_ *-*ForegroundColor Red

*throw* " Unable to gather end users.`r`n Check that User.Read.All scope is assigned and approved."

    }

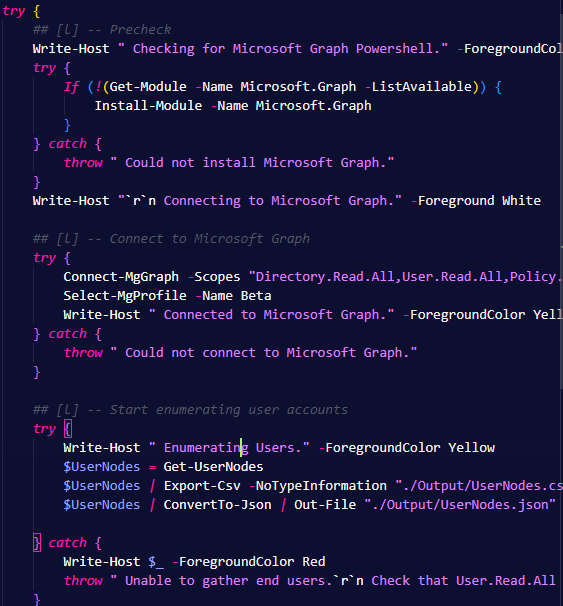
1. With this generic code block, we can start to evaluate end users by calling the Get-UserNodes function we defined in the NodeFunctions.psm1 file. To prove that it works, we’ll also add a couple exports (CSV and JSON) to review the data.

$UserNodes *=* Get-UserNodes

$UserNodes *|* Export-Csv *-*NoTypeInformation "./Output/UserNodes.csv"

$UserNodes *|* ConvertTo-Json *|* Out-File "./Output/UserNodes.json"

1. That it for this file. The main try block from the ps1 file should look like the image below.



1. Save all the files again and run the script like before. The output should appear like the image below.  
   Text

   Description automatically generated
2. With the script run, validate that the files in the Output folder align with the data found in your tenant.
3. Complete the steps 2 through 22 for all the following nodes:
   1. Roles
   2. Groups
   3. Conditional Access Policies
   4. Service Principals

## Associating user, groups, and roles

At this point, every run of the script should include all of nodes we need for basic analysis of Conditional Access Policies. In this section, we’re going to start connecting users and groups and roles which will be a bit different than other edges.

1. Open the EdgeFunctions.psm1 file in your editor.
2. For the edge functions, we need to take in arguments, so lets start with creating a new function with some boiler plate formatting.

*Function* Get-GroupMemberEdges() {

*Param*(

        $Groups *# Take in groups*

    )

    $Return *=* *@*() *# List of edges to be returned*

    $Count *=* 0    *# Count of groups analyzed*

*# Taking each group, get all group members*

*ForEach*($Group *in* $Groups) {

        $Count*+=*1

        Write-Progress *`*

*-*Id 1 *`*

*-*Activity "Connecting Group to Nodes" *`*

*-*Status $Group.DisplayName *`*

*-*PercentComplete ($Count*\**100 */* $Groups.Count)

*# Analyze each group membership below....*

    }

    Write-Progress *`*

*-*Id 1 *`*

*-*Activity "Connecting Group to Nodes." *`*

*-*Status "Complete" *`*

*-*Completed

*return* $Return

}

*Note: See how we take the groups and iterate through each group. We’ve included the progress bar logic we also used in the gathering node functions to keep the end user reprised of the status.*

1. Next, after the Write-Progress in the ForEach group, we need to take the group and get all the group members with the following code.

$GroupMembers *=* Get-MgGroupMember *-*GroupId $Group.Id

1. With each of the group members identified, we can create the edges. Edges are the unique ID from the source to the Unique ID from the Target and the relationship label. We then add the Edge to the Return list.

*ForEach*($GroupMember *in* $GroupMembers) {

    $Edge *=* [*BasicEdge*]::new(

        "MemberOf"*,* #Relationship

        $GroupMember.Id*,* #Source

        $Group.Id #Target

    )

    $Return *+=* $Edge

}

1. With this complete, we can now return all the edges to the main script. The completed function is pictured below.

Text

Description automatically generated

1. Open the **m365-to-cypher.ps1** file.
2. After the last node gathering code block within the primary try block, add in a new basic code block to map group members and output the edges as CSV and JSON.

*## [l] -- Mapping Groups*

*try* {

        Write-Host " Mapping groups." *-*ForegroundColor Yellow

        $GroupToAllEdges *=* Get-GroupToAllEdges *-*Groups $GroupNodes

        $GroupToAllEdges *|* Export-Csv *-*NoTypeInformation "./Output/GroupToAllEdges.csv"

        $GroupToAllEdges *|* ConvertTo-Json *|* Out-File "./Output/GroupToAllEdges.json"

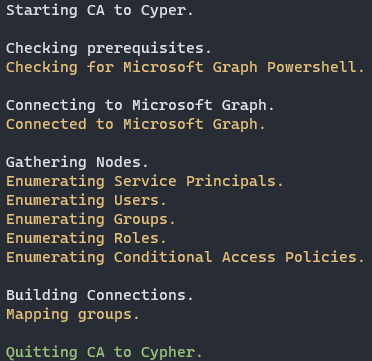
    } *catch* {

        Write-Host $\_ *-*ForegroundColor Red

*throw* " Could not map groups."

    }

1. Now try running the code. The output should align with the image below.



1. Repeat these instructions, but for roles instead of groups.

## Mapping everything to CA Policies

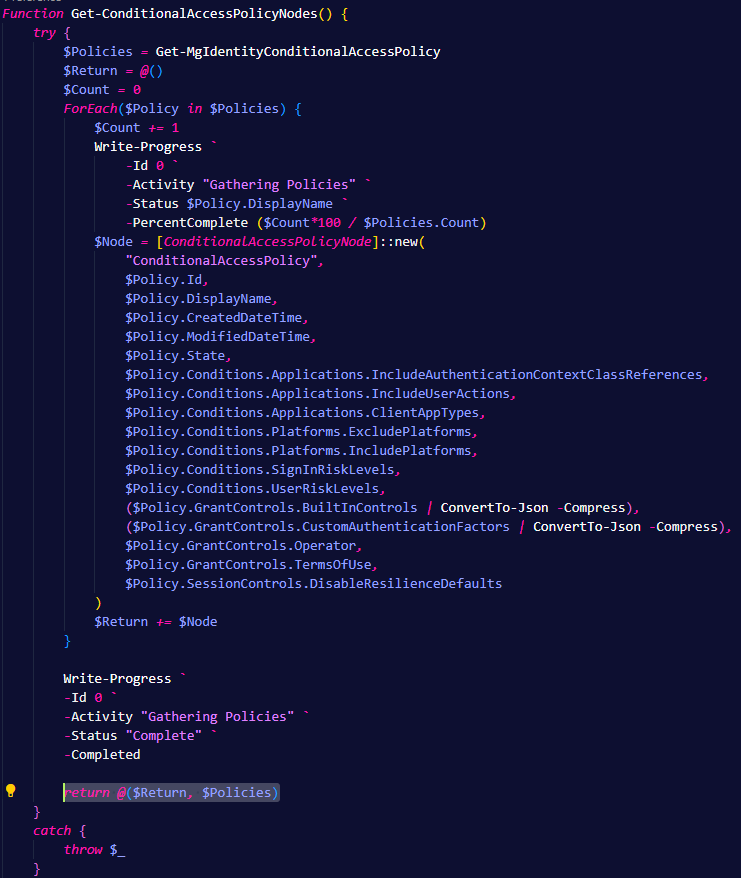
1. Open **NodeFunctions.psm1** and navigate to the Get-ConditionalAccessPolicyNodes function.
2. We’re going to modify the code so that it returns two values: a list of edges and the raw list of policies returned from Microsoft Graph. To do this, simply update the return statement with the following code.

*return* *@*($Return*,* $Policies)

1. Your code may look like the function below.

Text

Description automatically generated



1. Save this file and open **m365-to-cypher.ps1**.
2. Navigate to the code segment that gathers Conditional Access Policy Nodes. Replace it with the code below.

*## [l] -- Start enumerating ca policies*

*try* {

Write-Host " Enumerating Conditional Access Policies." *-*ForegroundColor Yellow

$CAPolicyInfo *=* Get-ConditionalAccessPolicyNodes

*# This is the final piece of node collection. All needed nodes should now exist.*

        $ConditionalAccessPolicyNodes *=* $CAPolicyInfo[0]

        $RawConditionalAccessPolicies *=* $CAPolicyInfo[1]

        $ConditionalAccessPolicyNodes *|* Export-Csv *-*NoTypeInformation "./Output/ConditionalAccessPolicyNodes.csv"

        $ConditionalAccessPolicyNodes *|* ConvertTo-Json *|* Out-File "./Output/ConditionalAccessPolicyNodes.json"

    } *catch* {

        Write-Host $\_ *-*ForegroundColor Red

*throw* " Unable to gather devices.`r`n Check that Policy.Read.All is assigned and approved."

    }

1. Now, we can break down CA relationships into three buckets:
   1. Protects (CA)->[Protects]->(App)
   2. Enforces (CA)->[Enforces]->(User|Group|Role)
   3. NotEnforces (CA)-[NotEnforces]->(User|Group|Role)
2. Open **EdgeFunctions.psm1** and lets create a function to Get-ApplicationToCaEdges that will take two parameters – Application and CA Policies. For this function, we want to pass in the RawConditionalAccessPolicies which contains the relationship information we need.

*Function* Get-ApplicationToCaEdges() {

*Param*(

        $CaPolicies*,*

        $Apps

    )

    $Return *=* *@*()

    $Count *=* 0

*ForEach*($CaPolicy *in* $CaPolicies) {

        $Count*+=*1

        Write-Progress *`*

*-*Id 1 *`*

*-*Activity "Connecting apps to policies" *`*

*-*Status $CaPolicy.DisplayName *`*

*-*PercentComplete ($Count*\**100 */* $CaPolicies.Count)

    }

    Write-Progress *`*

*-*Id 1 *`*

*-*Activity "Connecting apps to policies." *`*

*-*Status "Complete" *`*

*-*Completed

*return* $Return

}

1. This function will take each policy and evaluate the protected applications defined in the CA Policy. Since the CA policy stores the Unique APP Id, we don’t need to do any graph calls to make these correlations. Using the following code, we can ensure apps are specified in policy and act based on inclusion or exclusion.

*If* (    $CaPolicy.Conditions.Applications.IncludeApplications.count *-gt* 0 -or

                $CaPolicy.Conditions.Applications.ExcludeApplications.count *-gt* 0 ) {

*ForEach*($Included *in* $CaPolicy.Conditions.Applications.IncludeApplications) {

            }

*ForEach*($Excluded *in* $CaPolicy.Conditions.Applications.ExcludeApplications) {

         }

        }

1. For the Included applications, if “All” applications is selected in the policy, we need to Add All applications which are not going to be listed in the CA Policy. If All is not listed, then we can just use the IDs listed in the policy configuration with the code below.

*if* ($Included *-eq* "All") {

*ForEach*($App *in* $Apps) {

                        $Edge *=* [*BasicEdge*]::new(

                            "Protects"*,*

                            $CaPolicy.Id*,*

                            $App.Id

                        )

                        $Return *+=* $Edge

                    }

                } *else* {

                    $Edge *=* [*BasicEdge*]::new(

                        "Protects"*,*

                        $CaPolicy.Id*,*

                        $Included

                    )

                    $Return *+=* $Edge

                }

1. Similarly, we need to do the same for excluded.

*if* ($Excluded *-eq* "All") {

*ForEach*($App *in* $Apps) {

            $Edge *=* [*BasicEdge*]::new(

                "NotProtects"*,*

                $CaPolicy.Id*,*

                $App.Id

            )

            $Return *+=* $Edge

        }

    } *else* {

        $Edge *=* [*BasicEdge*]::new(

            "NotProtects"*,*

            $CaPolicy.Id*,*

            $Excluded

        )

        $Return *+=* $Edge

    }

1. The completed function is shown below.

Text

Description automatically generated

1. Save the file and open **m365-to-cypher.ps1**.
2. Use the code below to call this new function within the main try block after all the node gathering.

*## [l] -- Connecting CA Policies and Service Principals*

*try* {

        Write-Host " Connecting Applications to Policies." *-*ForegroundColor Yellow

        $SpCaEdges *=* Get-ApplicationToCaEdges *-*CaPolicies $RawConditionalAccessPolicies *-*Apps $ServicePrincipalNodes

        $SpCaEdges *|* Export-Csv *-*NoTypeInformation "./Output/ApplicationCaEdges.csv"

        $SpCaEdges *|* ConvertTo-Json *|* Out-File "./Output/ApplicationCaEdges.json"

    } *catch* {

        Write-Host $\_ *-*ForegroundColor Red

*throw* " Could not map applications to policies."

    }

1. Save and run the application. If all has gone well, you should have Json and CSV files mapping applications to CA policies.
2. Complete the previous steps for users, roles, and groups. The final output of the script should look like this.  
   Text

   Description automatically generated

## Converting the Data to Cypher

1. Open the **CyperFunctions.psm1** file.
2. The first function we need to create takes a list of nodes and exports them in cypher query for creation.

*Function* Export-Node {

*Param*(

        $Nodes*,*

        $Method

    )

*# Output string for the final file.*

    $Output *=* ""

*# Filter through the nodes.*

*ForEach*($Node *in* $Nodes) {

        $Label *=* $Node.Type

        $Values *=* $Node *|* ConvertTo-Json *-*Compress

*# Format it so that Neo doesn't get mad*

*# # This one removes the quotes around the property label*

        $Values *=* $Values *-replace* '"(\w\*)":'*,* '$1:'

        $Output *+=* "CREATE "

        $Output *+=* "(:$Label $Values);"

    }

*# Write the output*

*if* ($Method *-eq* "Append") {

        $Output *|* Out-File *-*Append ".\Output\GraphData.cypher"

    } *else* {

        $Output *|* Out-File ".\Output\GraphData.cypher"

    }

*return* $True

}

1. Following suit, we need another function to exported edges.

*Function* Export-Edge {

*Param*(

        $Edges*,*

        $Method

    )

*# Output string for the final file.*

    $Output *=* ""

*# Filter through the nodes.*

*ForEach*($Edge *in* $Edges) {

        $Label *=* $Edge.Relationship

        $Values *=* $Edge *|* ConvertTo-Json *-*Compress

*# Format it so that Neo doesn't get mad*

*# # This one removes the quotes around the property label*

        $Values *=* $Values *-replace* '"(\w\*)":'*,* '$1:'

        $Output *+=* "MATCH (n {Id:`"*$*($Edge.Source)`"}) "

        $Output *+=* "MATCH (x {Id:`"*$*($Edge.Target)`"}) "

        $Output *+=* "CREATE (n)-[:$Label]->(x);"

    }

*# Write the output*

*if* ($Method *-eq* "Append") {

        $Output *|* Out-File *-*Append ".\Output\GraphData.cypher"

    } *else* {

        $Output *|* Out-File ".\Output\GraphData.cypher"

    }

*return* $True

}

1. Finally, we need a function to do some small cleanup after the neo4j database is cleaned up. This resolves issues where a CA policy conflict arises, making it looks like users|groups|roles are enforce AND not enforced at the same time.

*Function* Append-CleanUp {

    $Output *=* ""

    $Output *+=* "MATCH (u)-[a:Enforces]->(b)<-[s:NotEnforces]-(u) DELETE a;"

*# Write the output*

*if* ($Method *-eq* "Append") {

        $Output *|* Out-File *-*Append ".\Output\GraphData.cypher"

    } *else* {

        $Output *|* Out-File ".\Output\GraphData.cypher"

    }

}

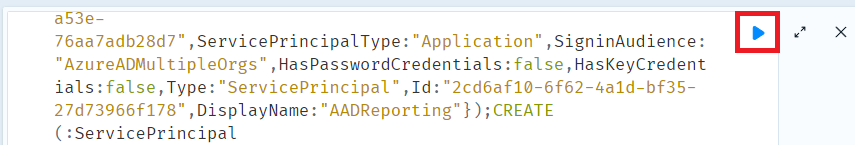
1. Save the file and close. Open the **m365-to-cypher.ps1** file.
2. Navigate to the end of the try block and start adding in the export functions. Start by exporting all the nodes and then exporting the edges. Finally, export the cleanup function.

Text

Description automatically generated

1. Save the file and run the code again. All the nodes and edges should be output to the **Output/GraphData.cypher** file.

## Import the Graph into Neo4j

1. Open the **GraphData.cypher** file in your preferred text editor.
2. Open Neo4j and connect to the project database as we did in Lab 3.
3. In the query box, copy and paste the complete contents of the **GraphData.cypher** file into the query box and press the run button.  
   
4. Wait. This might take a while. Like, seriously, good time to grab a refreshing work appropriate drink.
5. Return your results in neo4j with the following query  
   MATCH (A) RETURN (A)

Scatter chart

Description automatically generated