Hands-On Lab

Exploring the HPE DSCC Rest API using PowerShell

## Requirements:

1. You will need access to a machine that can open and run a PowerShell prompt which can access the internet.
2. You will need the HPEDSCC.Zip file which contains a prototype PowerShell Module for the HPE DSCC environment, this file should be located on your desktop. This can be downloaded directly from the Github site at; https://github.com/HewlettPackard/HPEDSCC-PowerShell-Toolkit
3. You will need either a Client ID & Client Secret Combination or an Access Token for the connection. You will be supplied with access to a share which contains the Access Token in a text file for use in the lab.

## Disclaimer:

The provided HPEDSCC PowerShell Module Prototype is not an HPE Product, and as such not supported by HPE in any way. This PowerShell Module project is ONLY used to help in the discovery of the HPE DSCC API, and makes exploring the model easily done from a PowerShell Command Line.

## Access Tokens

Additionally, the Access Token will be created 10 minutes before the lab starts, and we will not be distributing the Client ID or Client Secret as this is a shared environment using a single account. If we were to distribute the Client ID and Client Secret, each time a new PowerShell connection was used to generate a new Access Token, it would immediately invalidate the Access Token supplied to the previous user. The Access Token should remain active for 2 hours, which will cover the duration of the Lab.

## Commands you type

In all of the following examples of PowerShell windows, the things you are expected to type are both Bold and Dark Black, while the information returned is shown in a lighter grey color to signify it is output as opposed to your input.

Additionally, your PowerShell prompt will appear as below;

PS C:/users/{myusername}/desktop/HPEDSCC>

However to make the commands more read-able, I have truncated the prompt to simply

PS:>

I also will show what you type in BOLD versus the output expected from the command.

## Scope

The commands included in this PowerShell Module are compatible with PowerShell on windows, however no commands require any features of functions (such as .NET6) that won’t also work on either a PowerShell prompt on either a Linux or Mac, but this lab does not include the process to install PowerShell on those platforms.

### Open Source

The entire toolkit is written completely open source, and is self-documented. You are welcome to open any of the files and examine the source code as none of it is compiled in any way. I recommend installing VSCode (Free download) tool, as it’s a very good open source editor that understands PowerShell. Each command relates to a single Function defined in the various files in the subfolder called ‘Scripts’

# Step 1: Setting Up Your Environment;

The PowerShell Module Zip file should be extracted to the desktop into a folder with the same name. Additionally, you will want to copy the AccessToken.txt file from the Share to your local desktop as well.

Once these steps have been done, we need to open a new PowerShell Window. To do this simply hit the Windows key on the Keyboard, and type ‘PowerShell’ and hit enter. This will open a standard PowerShell window. Your present working directory will likely be ‘C:\Users\{yourUserName}.

Change directory to the folder created previously for the PowerShell Toolkit.

PS:> **CD Desktop**

PS:> **CD HPEDSCC**

If you issue a ‘DIR’ command, your PowerShell window should look like this.

PS:> **dir**

Directory: C:\Users\lionettc\desktop\HPEDSCC

Mode LastWriteTime Length Name

---- ------------- ------ ----

d----- 11/5/2021 1:26 PM scripts

-a---- 11/4/2021 5:01 PM 652 HPEDSCC.psd1

-a---- 11/5/2021 1:55 PM 2124 HPEDSCC.psm1

-a---- 11/4/2021 3:40 PM 6363 HPEDSSC.format.ps1xml

-a---- 9/13/2021 7:24 AM 1546 LICENSE

-a---- 10/5/2021 6:24 PM 3124 README.md

# Step 2 Importing (loading) and understanding the Module

You can get a list of currently loaded modules with the command;

PS:> **Get-Module**

And you will notice that a few Modules may already be loaded by the Operating System. To load the HPEDSCC Module use the following command;

PS C:\Users\lionettc\desktop\HPEDSCC> **Import-Module .\HPEDSCC.psm1**

WARNING: This a Prototype PowerShell Module, and not supported by HPE.

Once the module is loaded, you can validate this using the same Get-Module command listed before, and you will see the Module; To obtain a list of valid commands that are included with this module, use the following command;

PS:> **Get-Command -Module HPEDSCC**

CommandType Name Version Source

----------- ---- ------- ------

Function Connect-HPEDSCC 0.0 HPEDSCC

Function Get-DSCCAlertOrContact 0.0 HPEDSCC

Function Get-DSCCCapacity 0.0 HPEDSCC

Function Get-DSCCCapacityHistory 0.0 HPEDSCC

Function Get-DSCCCapacitySummary 0.0 HPEDSCC

Function Get-DSCCCertificate 0.0 HPEDSCC

Function Get-DSCCComponentPerfStats 0.0 HPEDSCC

Function Get-DSCCController 0.0 HPEDSCC

Function Get-DSCCControllerSubComponent 0.0 HPEDSCC

Function Get-DSCCDisk 0.0 HPEDSCC

Function Get-DSCCEvent 0.0 HPEDSCC

Function Get-DSCCHost 0.0 HPEDSCC

Function Get-DSCCHostGroup 0.0 HPEDSCC

Function Get-DSCCVolume 0.0 HPEDSCC

Function Get-DSCCInitiator 0.0 HPEDSCC

Function Get-DSCCMail 0.0 HPEDSCC

Function Get-DSCCShelf 0.0 HPEDSCC

Function Get-DSCCStoragePool 0.0 HPEDSCC

Function Get-DSCCStoragePoolVolume 0.0 HPEDSCC

Function Get-DSCCStorageSystem 0.0 HPEDSCC

Function Get-DSCCVolume 0.0 HPEDSCC

Function Invoke-DSCCControllerLocatePCBM 0.0 HPEDSCC

Function Invoke-DSCCShelfLocate 0.0 HPEDSCC

Function Invoke-DSCCStorageSystemLocate 0.0 HPEDSCC

Function New-DSCCAlertContact 0.0 HPEDSCC

Function New-DSCCHost 0.0 HPEDSCC

Function New-DSCCHostGroup 0.0 HPEDSCC

Function New-DSCCInitiator 0.0 HPEDSCC

Function New-DSCCMail 0.0 HPEDSCC

Function Remove-DSCCAlertOrContact 0.0 HPEDSCC

Function Remove-DSCCHost 0.0 HPEDSCC

Function Remove-DSCCHostGroup 0.0 HPEDSCC

Function Remove-DSCCInitiator 0.0 HPEDSCC

Function Remove-DSCCMail 0.0 HPEDSCC

Function Set-DSCCAlertContact 0.0 HPEDSCC

Function Set-DSCCHost 0.0 HPEDSCC

Function Set-DSCCHostGroup 0.0 HPEDSCC

Once you have a list of commands, you can get help on any of the commands using the following PowerShell Commends; Try these out to get an idea of how to obtain various types of help.

PS:> **Get-Help Connect-HPEDSCC**

PS:> **Get-Help Connect-HPEDSCC -examples**

PS:> **Get-Help Connect-HPEDSCC -detailed**

PS:> **Get-Help Connect-HPEDSCC –full**

The standard Help is a summary help, and includes the Syntax, The example of this output is below;

PS:> **get-help Connect-HPEDSCC**

NAME

Connect-HPEDSCC

SYNOPSIS

Connects to an HPE DSCC Endpoint.

SYNTAX

Connect-HPEDSCC -Client\_Id <String> -Client\_Secret <String> -GreenlakeType <String>

[-WhatIf] [<CommonParameters>]

Connect-HPEDSCC -AccessToken <String> -GreenlakeType <String> [-WhatIf] [<CommonParameters>]

DESCRIPTION

Connect-HPEDSCC is an advanced function that provides the initial connection to a HPE Data

Storage Cloud Services

This summary contains one large clue however in the Syntax to show how the command works. Note that there are two ways this command can be used, and they are mutually exclusive, you can either supply the Client\_ID and Client\_Secret, OR you can supply the AccessToken. Additionally, you will notice that options such as –WhatIf are contained within square brackets. This indicates that this is an optional parameter. From this you can also determine that arguments not contained within square brackets are required parameters.

To see examples of how each command can be used, you can use the second option by adding the –examples to the Get-Help command.

This can guide you to how a command works, but it may lack the details about what acceptable values are for each field. To obtain the information about what each parameter means, you can use the –detailed option. Note that the –detailed option also includes the examples. The section that the detailed option gives you in addition that helps you determine the values for each parameter are in the PARAMETERS section.

PARAMETERS

-Client\_Id <String>

The Client ID that is given to the user from the DSCC service. This can be found on the

DSCC GUI. If the Client ID is specified, so must the Client\_Secret.

-Client\_Secret <String>

The Client Secret which is given when a new API account is created, this item must be remembered from when the account was created, and is not available anywhere in the DSCC gui. If the Client\_Secret is specified the Client\_Id must also be provided.

-AccessToken <String>

If the Access Token can be gathered seperately, you can use this to authenticate your

RestAPI Calls.

If no AccessToken is specified a Client\_Id or Client\_Secret need be specified.

-GreenlakeType <String>

This can either be the Production Instance of DSSC or it can be the Region/Dev instance of the DSSC. The values can either be Dev or the various regions such as US, EU, or Asia

-WhatIf [<SwitchParameter>]

This option shows you the command that will be sent to the DSCC, will include the URI being sent to, the Header, Method, and the Body of the message.

<CommonParameters>

This cmdlet supports the common parameters: Verbose, Debug, ErrorAction, ErrorVariable, WarningAction, WarningVariable, OutBuffer, PipelineVariable, and OutVariable. For more information, see about\_CommonParameters (https:/go.microsoft.com/fwlink/?LinkID=113216).

Spend a few minutes getting help on other commands that interest you.

# Step 3: How to use AutoComplete/AutoFill

There are a number of shortcuts designed in PowerShell to assist you in being productive. The first of these is the autocomplete feature. As an example, lets get a list of valid commands

PS:> **Get-Com**

At this point, hit <TAB> and you will see PowerShell will attempt to fill out the name of the command with the rest of the word. Now to add parameters, we want to select the –module parameters, but again, we can just type the first letter and <TAB> and it will fill in the rest of the word for us. You can then start typing the name of the module but one you have a letter or two, hit tab again, and it will attempt to autofill the field for you.

Below is the example of what I type, and below that is what appears on the screen.

PS:> **Get-Com<TAB> -m<TAB> hp<TAB>**

PS:> **Get-Command –module HPEDSCC**

What happens if the autofill puts in the wrong value? That is common when multiple possible completions exist, but by hitting tab multiple times you can walk through the different valid options until the correct one appears. As an example of this, let’s use <TAB> to walk through all of the possible HPE Commands that start with New-HPE

PS:> **New-HP<TAB><TAB><TAB><TAB>**

Once you have exhausted the list, it will cycle back to the first option again. This is another way to discover commands.

You can also use this method to discovery or cycle through the options of a command such as New-HPEDSCCDOMHostServiceHost. You can simply add the ‘-‘ and type tab, which will cycle you through the possible parameters. Once a parameter has been added, it will no longer appear in the cycle of autocompletes.

PS:> **New-DSCCHost -<TAB>**

Some parameters cannot be auto-filled however, since things like –VolumeName can be any string that you want to enter, however when a parameter has only a limited number. An example of this is if you use the following –OperatingSystem Parameter and type a <space> and then <TAB> again, and you will find that you can toggle through all of the supported operating systems; TAB through these until you cycle back the first one to ensure that you have seen the entire list of supported Operating Systems.

PS:> **New-DSCCHost –OperatingSystem <TAB><TAB><TAB><TAB>**

Special note here; some values use a single quote around them, while other don’t. This is due to the fact that when PowerShell detects a space, it assumes the next thing in the next argument. To use a string that contains spaces (or other special characters) the single quotes ensure that the string is one value. This is most common on things like –comment which might contain a sentence like ‘This Volume is used as scratch space for SQL23’

This section is presented to lower the amount of repetitive typing that you have to do since all of the command names are long.

# Step 4: How to use the –WhatIf option

Each command in the toolkit contains the option called –WhatIf, which will let you interrogate what command would be sent to DSCC to satisfy the list of parameters given. This will include the Rest API Endpoint (HTTPS location) that the command is sent, the Header (which includes the Access Token), the Method used (Put, Post, Delete, Get) and the body (if one exists) for the command;

To show you how this works, lets combine what we have learn use the following command; Use the Get-Command option to get a list of valid commands, select your favorite command that starts with ‘GET’, and from the command line. Use the Get-Help to determine if that command has a required parameters. If parameters are required, simply use dummy data like ‘12345’ or ‘6789’. If there exist non-required parameters, leave those unset

PS:> **Get-DSCCCertificate –SystemID 12345 –whatIf**

PS C:\Users\lionettc\desktop\HPEDSCC> Get-DSCCCertificate -SystemID 1234 -whatIf

WARNING: You have selected the What-IF option, so the call will note be made to the array,

Instead you will see a preview of the RestAPI call

The URI for this call will be

storage-systems/device-type1/1234/certificates

The Method of this call will be

Get

The Header for this call will be :

The Body of this call will be:

{

}

You will first notice that the Header is blank. If this command were run against DSCC, you would get an access error since you lack authentication. When we run this same command after we connect to the array a header will exist.

The second thing to notice is that the URL to connect to is shown as storage-systems/device-type1/1234/certificates which is incomplete as well. Do notice however that the partial URL that is included, has the parameter that you passed incorporated into the request.

We will revisit this once we have connected to the array.

# Step 5: Connecting to the HPE DSCC

Now to connect to the HPE DSCC endpoint, we will need to use the following command. This connection command basically populates local variables in PowerShell that enable all of the other commands to function, such as the default header.

You will need to copy the complete Access Token listed in the AccessToken.txt file on the Lab share (or posted the Github site right before the HOL) into the clipboard. Once you have, use the following command;

PS:> Connect-HPEDSCC –AccessToken <RIGHT-CLICK> -GreenlakeType Dev

When you right click in the PowerShell window, it will insert the Access Token from the Keyboard. Your final command will appear similar to this.

PS:> **Connect-HPEDSCC –AccessToken eyJhbGciOiJSUzI1NiIsImtpZCI6InpDb3RUbnIybUUycEFHSWZRVlctWEY5QjRCVSIsInBpLmF0bSI6ImRlejAifQ..VzuHt4gEz9E7AV\_fa3lspnZkjBzacY1z5iiMwHOHfSaGOobalHwFhIka5W8FvyG36Fq2Lrc3vCbBKYOGLJw7e2xRR7HsD92W2QjLGQqRkgrLQjgqavae0xddbW2uClCpxCKQENZwp\_124TEgirSY\_uaiLD3GdwBBF6a0XNU5KytK3sKqM5OiwgnON4YnJwYZuv2mMf4af8FbNgyyNZ0dxDFwG9Pvt9dx6w4JIz4L2veLk2bVHj0A6XyZTJTdQs4qKvHlXwPfo4fos5SBz3qmkQi096Ew6Xds7KY\_RE5qy\_eEcyueJukuJUkp9VJSGIsQHfQJFKBvtvBF8GgFkaEuw**

**-GreenlakeType Dev**

Name Value

---- -----

Access\_Token eyJhbGciOiJSUzI1NiIsImtpZCI6InpDb3RUbnIybUUycEFHSWZRVlctWEY5QjRC0..

If we repeat the command from Step 4, exploring the value from the WhatIf Option, you will see that both the Header and the URL are now complete, since you have identified the Access Token as well as the GreenLake type.

PS:> **Get-DSCCCertificate -SystemID 1234 -whatIf**

WARNING: You have selected the What-IF option, so the call will note be made to the array,

instead you will see a preview of the RestAPI call

The URI for this call will be

https://scalpha-app.qa.cds.hpe.com/api/v1/storage-systems/device-type1/1234/certificates

The Method of this call will be

Get

The Header for this call will be :

{

"Authorization": "Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6InpDb3RUbnIybUUycEFHSWZRVlctWEY5QjRCVSIsInBpLmF0bSI6ImRlejAifQ..VzuHt4gEz9E7AV\_fa3lspnZkjBzacY1z5iiMwHOHfSaGOobalHwFhIka5W8FvyG36Fq2Lrc3vCbBKYOGLJw7e2xRR7HsD92W2QjLGQqRkgrLQjgqavae0xddbW2uClCpxCKQENZwp\_124TEgirSY\_uaiLD3GdwBBF6a0XNU5KytK3sKqM5OiwgnON4YnJwYZuv2mMf4af8FbNgyyNZ0dxDFwG9Pvt9dx6w4JIz4L2veLk2bVHj0A6XyZTJTdQs4qKvHlXwPfo4fos5SBz3qmkQi096Ew6Xds7KY\_RE5qy\_eEcyueJukuJUkp9VJSGIsQHfQJFKBvtvBF8GgFkaEuw"

}

The Body of this call will be:

{

}

# Step 6: Exploring the RestAPI Calls

We should now be able to using the WhatIf option discover the exact RestAPI calls to accomplish specific tasks. First select a command that should return a list of storage systems. Using the Get-Help command we can see that we are required to present a –DeviceType and using <TAB>-complete we can see what the valid options are. In this case DeviceType1 refers to 3PAR/Primera/Alletra9K, and DeviceType2 refers to Nimble/Alletra6K.

Use the following command;

PS: > **Get-DSCCStorageSystem -DeviceType device-type1 -WhatIf**

From the output of this command you should be able to determine the URL, the contents of the header, the contents of the body and the Method of the HTTPS call. Note: This information can be transcribed directly into a Linux based CURL command, or any other language to use as well.

Now we should examine the effects of certain variable that are imbedded in the URL for very specific calls. As our example, we will use the follow ‘Remove’ type command;

PS:> **Remove-DSCCHost –HostId 12345 –WhatIf**

The output of this will show you how the parameter HostId and its value of 12345 is inserted into the URL so that the RestAPI endpoint knows what object you are operating on. Additionally you should be able to determine that the call is using the ‘Delete’ method, which informs the RestAPI endpoint what type of activation should be done to the object.

Now that we have seen how the Parameters may be inserted into the URL, or the type of command is expressed in the HTTPS Method (Delete), we should discuss the last set of operations such as New and Set, which will allow you to create new objects or modify existing objects on the Rest endpoint.

Using the following command, add as many of the parameters as you can, You can use dummy values for most of them.

PS: > **New-DSCCAlertContact -SystemID 12345 -company MyCompany -companyCode 42 -country USA -fax 4259490000 -firstName Chris -lastName Lionetti -includeSvcAlerts $true -notificationServices $true -preferredLanguages English -primaryEmail chris.lionetti@hpe.com -primaryPhone 425-949-1111 -receiveEmail chris.lionetti@hpe.com -receiveGrouped admin@hpe.com -secondaryEmail donthaveone@null.com -secondaryPhone 425-555-1212 -whatIf**

Note that for items that are True and False, the proper value to insert the Boolean value is ‘$true’ or ‘$false’.

This time you will notice that the body of the message contains a most of these parameters.

# Step 7: Exploring the Objects that come back from the RESTAPI

We are now going to explore the object sets that are returned from the RestAPI.

If you look at some of the commands, such as Get-DSCCShelf there are required parameters such as SystemID. So without first discovering a valid system ID we cannot run this command. Generally a good place to start is with the command Get-DSCCStorageSystem. The only Parameter that is required for this command is the Device-Type command, and it will Auto-Fill for you. Run this command to see what the return object looks like.

PS:> **Get-DSCCStorageSystem -DeviceType device-type1**

You can scroll up and see the raw data in the object that is presented back. You should note that this data is presented on screen in PowerShell Object Mode. To display that same information in JSON native format, you can pipe the output of the command into the built-in JSON converter as such. Run the following command to see the differences.

PS:> **Get-DSCCStorageSystem -DeviceType device-type1 | convertTo-json**

You will notice that the amount of information returned is massive, and in this case represents multiple arrays. Since what is returned is a full PowerShell object you can select ONLY the first array in the list by adding a ‘(‘ to the start, and a ‘)’ to the end, and [#] where the number represents what element you want.

PS:> **(Get-DSCCStorageSystem -DeviceType device-type1)[1]**

PS:> **(Get-DSCCStorageSystem -DeviceType device-type1)[3] | convertTo-json**

Additionally you can choose to show the data in a table format, which makes discerning the information far easier.

PS:> **Get-DSCCStorageSystem -DeviceType device-type1 | format-table**

If you want to only show some fields on the table, you can force only those fields to display by name

PS:> **Get-DSCCStorageSystem –DeviceType –device-type1 | format-table id,type,name,systemwwn,deviceid,fqdn**

Now we can easily see the Storage System ID, we can make calls that require that piece of information. As an example, let’s retrieve the shelf information for one storage system. Make sure that you run the above command, and then highlight with your mouse one of the systems IDs….like system 2M2042059T and hit <CTRL>-C to copy this to the clipboard. When you run the next command, after you put in the –StorageSystemID you can hit space and them right click in the right place and it will insert that system ID so that you don’t have to type it. This is not very useful when the ID is only 8 characters long, but for some things the ID may be a 42 character hexadecimal number.

PS : > Get-DSCCShelf -StorageSystemId 2M2042059T -DeviceType device-type1

Now lets grab more detailed information about this item. You will see that some returned values are complex types, and the output has symbols like @{ or @( in various places. This indicates that the value in that field can be expanded. To dig into that field, you can do the same trick used before; namely, add a ‘(‘ at the start, a ‘)’ at the end, and then a ‘.’ And the name of the field to expand. Example, if one of the items in the returned data looks like this;

associatedLinks : {@{type=systems; resourceUri=/api/v1/storage-systems/device-type1/2M2042059T},@{type=enclosure-card-ports; resourceUri=/api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a282fc6ea8d81549e77dca70c/enclosure-card-ports}, @{type=enclosure-disks; resourceUri=/api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a282fc6ea8d81549e77dca70c/enclosure-disks}, @{type=enclosure-sleds; resourceUri=/api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a282fc6ea8d81549e77dca70c/enclosure-sleds}...}

If you were to modify the above command like this;

PS:> **(Get-DSCCShelf -StorageSystemId 2M2042059T -DeviceType device-type1).associatedLinks**

type resourceUri

---- -----------

systems /api/v1/storage-systems/device-type1/2M2042059T

enclosure-card-ports /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb177dca70c/enclosure-c...

enclosure-disks /api/v1/storage-systems/devictype1/2M2042059T/enclosures/0929b49e77dca70c/enclosure-d...

enclosure-sleds /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb70c/enclosure-s...

enclosure-fans /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929b282a70c/enclosure-fans

enclosure-expanders /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a28ca70c/enclosure-e...

enclosure-powers /api/v1/storage-systems/device-type1/2M2042059T/enclosures/092549e77dca70c/enclosure-p...

enclosure-cards /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bbe77dca70c/enclosure-c...

disks /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a282fc6edca70c/disks

And again, the above list is an array of items, so I can retrieve any of this by doing the same trick again.

PS:> **((Get-DSCCShelf -StorageSystemId 2M2042059T).associatedLinks)[2]**

type resourceUri

---- -----------

enclosure-fans /api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a28ca70c/enclosure-fans

And you can again dig deeper into this object to grab the ResourceURI, like this.

PS: > **(((Get-DSCCShelf -StorageSystemId 2M2042059T).assoc**

**iatedLinks)[4]).resourceUri**

/api/v1/storage-systems/device-type1/2M2042059T/enclosures/0929bb1a282fc6ea8d81549e77dca70c/enclosure-fans

Now lets try and grab the list of disks. Note that the Get-HPEDSCCDomDisks command currently only seems to work against the Nimble/Alletra6K type targets, so use Type-2 only. Note also that getting the Storage Systems for Type2, the IDs are far longer, so you will need to use clipboard method.

PS:> **Get-DSCCStorageSystem –DeviceType –device-type2 | format-table**

PS:> **Get-DSCCDisk -StorageSystemId 000849204632ec0d70000000000000000000000001 -De**

**viceType device-type2**

Again, the list will be very large, so look at the last object and determine what items you want displayed, and issue a command with just the relavent data;

PS:> **Get-DSCCDisk -StorageSystemId 000849204632ec0d70000000000000000000000001 -De**

**viceType device-type2 | format-table array\_name,model,shelf\_location,disk\_type,state,size**

# Step 9: Free Play

Free Play time. At this point, you should be a second hard with exploring objects. You should also discover that since this is a development system, some commands only work against a Type1 or Type2 Storage Systems, some arrays (specifically those that are virtual) don’t actually have resources to return. These are after-all Development Systems, and this PowerShell toolkit is far from complete. If you get a return that looks like the following;

PS:> **Get-DSCCController -StorageSystemId 2FF70002AC025F89 -DeviceType device-type1**

items : {}

total : 0

pageLimit : 320

pageOffset : 0

requestUri : https://scalpha-app.qa.cds.hpe.com/api/v1/storage-systems/device-type1/2FF70002AC025F89/nodes

In this case, DSCC accepted the call, but has no items to return.

# Step 10: Clean Up

To clean up the environment, once you have closed the PowerShell window, the variables are removed, If you open a new PowerShell Window, those connection variables are no longer accessible in that new window.

To remove the PowerShell Toolkit, simply delete the folder, there is no installation or de-installation process, and no changes to the server or workstation are made.