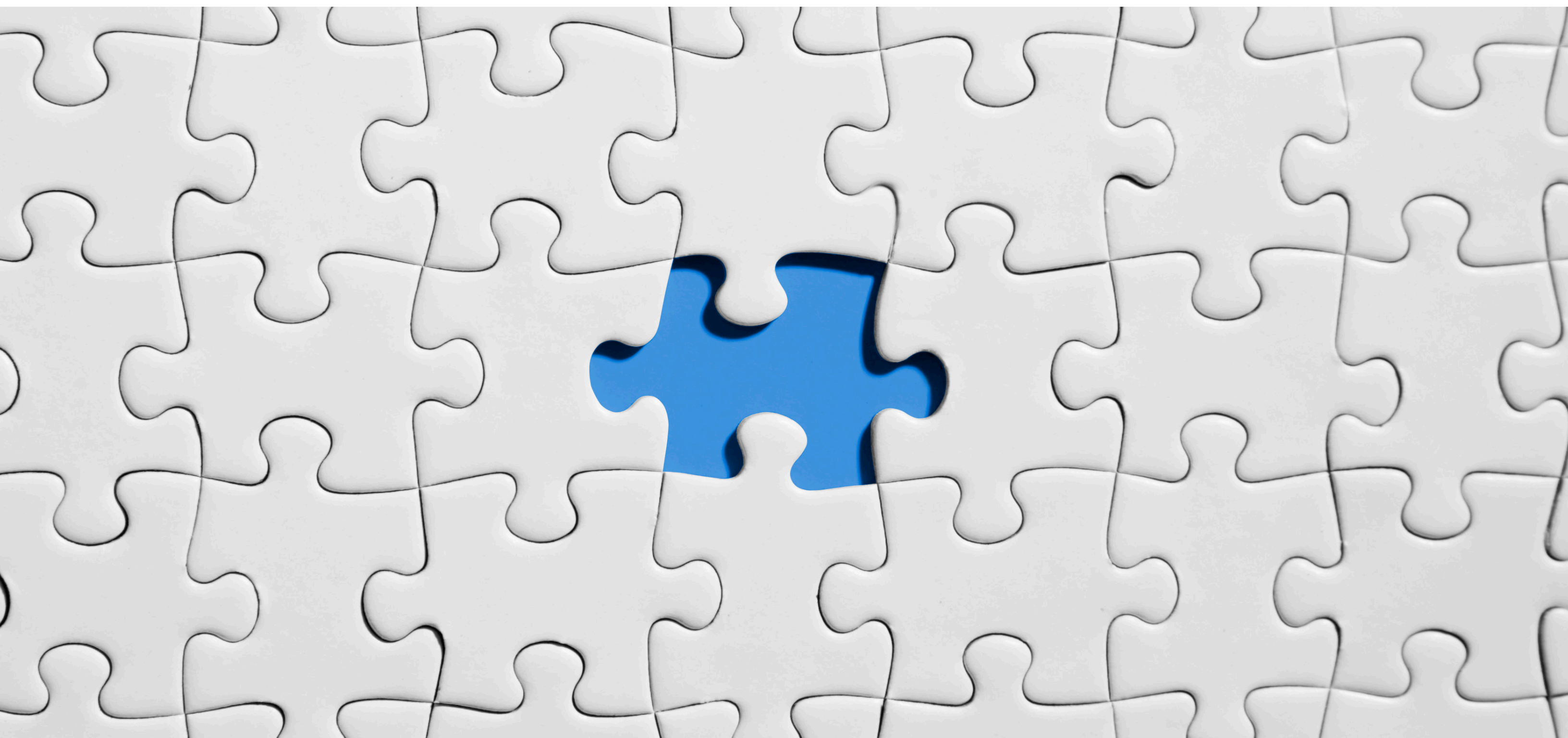
DELL EMC

Managing Unity Arrays with New Generation Tools



Samuel Thomas
samuel.thomas@hpe.com

Vijayakumar Ravindran
vijayakumar.ravindran@emc.com

DELL EMC PROVEN PROFESSIONAL

Knowledge Sharing Article
© 2017 Dell Inc. or its subsidiaries.

Table of Contents

Table of Figures 3

Introduction 4

Challenges 5

Solution 5

Unity REST API 6

PowerShell 7

PowerShell Modules 7

Integration of PowerShell Modules with Unity 8

Prerequisites 8

How to install PowerShell Modules 8

Working with Unity from PowerShell 9

Unity Administration Day 2 Task 12

Creating NAS Server 12

Output 15

Creating CIFS Server 15

Output 18

Summary 19

Bibliography 19

Disclaimer: The views, processes or methodologies published in this article are those of the authors. They do not necessarily reflect Dell EMC's views, processes or methodologies.

Table of Figures

Figure 1: Integration of Dell EMC Unity in Virtualized Environment 4
Figure 2: Dell EMC Unity Management with REST API 6
Figure 3: PowerShell version 8
Figure 4: Installing the PowerShell Modules 9
Figure 5: List of PowerShell Modlues 9
Figure 6: Unity Connectivity 11
Figure 7: List of Unity arrays connected 11
Figure 8: NAS Server creation with PowerShell Module 15
Figure 9: CIFS Server Creation with PowerShell Module 18

Introduction

Things change frequently in the modern IT world. Virtualized data centers have become denser and more complex creating performance bottlenecks, SLA risks, and management headaches. From a storage perspective, those changes result in complexity in deployment and management with highly skilled resources. Storage infrastructure plays a vital role in success of these demanding virtual and physical environments.

To meet these challenges Dell EMC has introduced Unity, a unified solution with the salient features required for modern IT, namely; Simplicity, Modern design, affordable price, and flexible deployment. Unity combines the power of VNX and simplicity of VNXe. It is available in Hybrid array, All-Flash array and Unity VSA (Software Defined Storage) which supports REST API. It boosts infrastructure and service agility with support for broad virtualization APIs including VAAI, VASA, VVol, and ODX. Automate infrastructure and enforce policy-based compliance. Virtualization with VMware/Microsoft and Dell EMC Unity Storage lets us modernize with agility and performance.

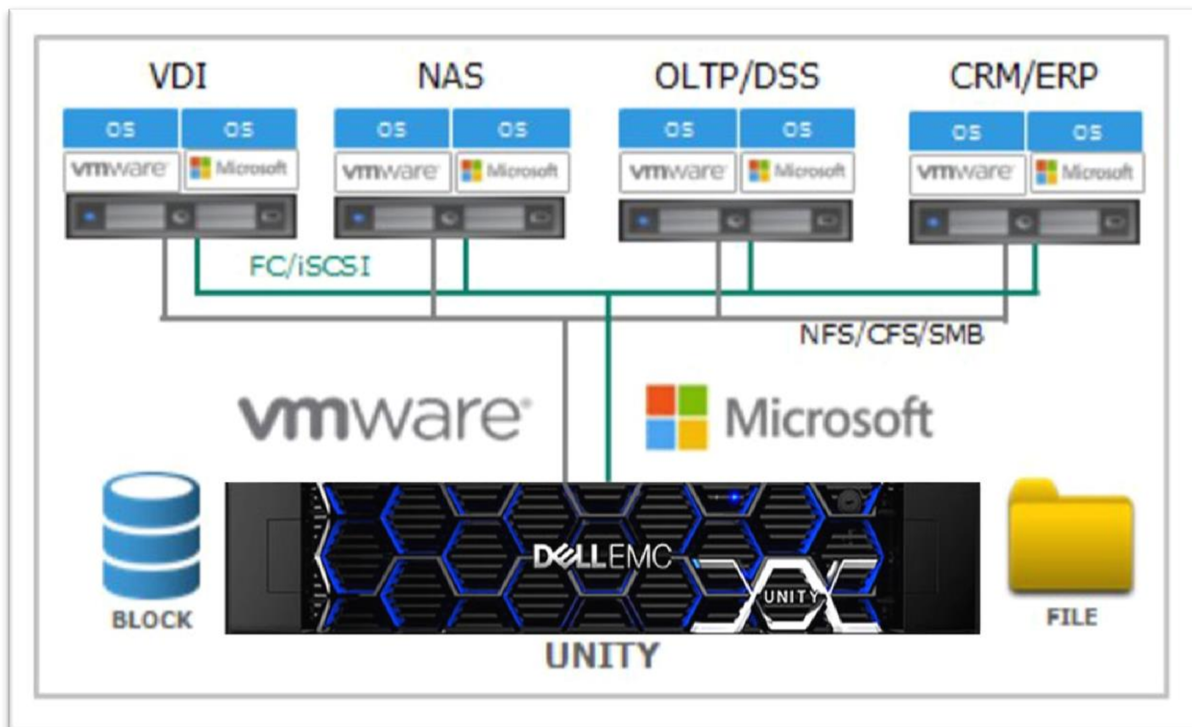


Figure 1: Integration of Dell EMC Unity in Virtualized Environment

This article illustrates integration of PowerShell modules with Unity REST API to automate and manage Unity arrays with minimal knowledge of automation and storage. Unity VSA has been used in the experiment to illustrate key research findings, with the ultimate goal of automating and managing Unity arrays using PowerShell modules.

This article will help IT administrators, storage architects, partners, Dell EMC employees and any other individuals enhance the management of Unity Arrays with basic knowledge about

storage and PowerShell. It also helps to understand Unity REST API unique features and how they can be integrated with procedural programming or script language which enables you (reader) to customize and develop on your own tool based on organization requirement. This article also shares the Day 2 operation of all Unity arrays in an effortless way by any IT admins.

Challenges

When installing arrays in the customer environment, a Customer Engineer/Field Engineer, does many facilitation tasks before hand-over of the array to the Operation team. Operation Team shall use our Dell EMCs SDDC platform, i.e. ViPR. Much of the time spent on tasks to perform platform readiness for the storage array and for automation and management depends on tools available in market.

Solution

Using Dell EMC Unity's REST API features, we overcame the challenges in terms of unique features such as Simple, Modernize, affordable price, and deployment flexibility. Unity REST API helps IT professionals, Developers, and Architects integrate procedural programming language or scripts with Unity systems to reduce manual intervention for performing regular tasks on the storage system.

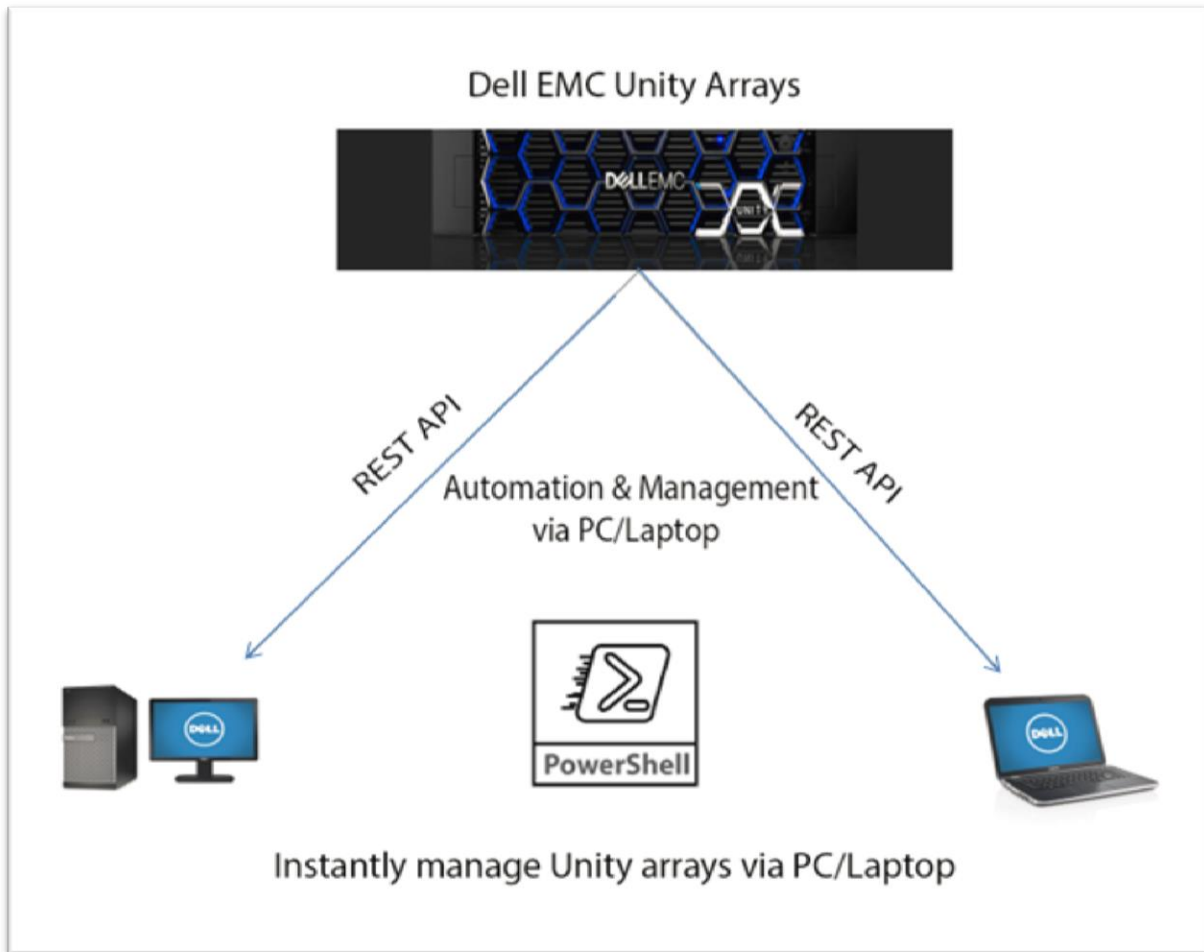


Figure 2: Dell EMC Unity Management with REST API

Unity REST API

Representational State Transfer (REST) is a common approach in today's IT management products and a frequent choice for many web-based APIs. REST API is modeled after Service-Oriented Architecture (SOA), but now over shadowed in terms of usage. Cloud computing and Micro services are almost RESTful APIs, which would rule the future. Thus, it is important for IT Professionals, Developers and Architects to make the most of REST because of it is simplicity and agility for new applications.

Though REST is client-server protocol, it totally separates the user interface from Server and Data Storage. For complex interactions, clients can use any procedural programming language, such as C++ or Java, or scripting language, such as Perl, Python or PowerShell, to make calls to the REST API.

By referencing this article, we can do the following operations on Unity System with the help of REST API.

- Configure system settings for the Unity storage system.

- Manage the connections to remote systems, including manage host configurations, iSCSI initiators, and iSCSI CHAP accounts.
- Configure network communication, including manage and create NAS Servers and set up iSNS for iSCSI storage.
- Manage storage, including configure storage pools and manage file systems, iSCSI, VMware, and Hyper-V storage resources.
- Protect data, including manage snapshots and replication sessions.
- Manage events and alerts.
- Service the system, including change the service password, manage Dell EMC Secure
- Remote Support (ESRS) settings, and browse service contract and technical advisory information.

PowerShell

This article explains each terminology of PowerShell with sample script and how we can use to integrate with REST API to manage Unity arrays, so it helps you to customize your script depends on you or your organization requirement. As discussed earlier we can integrate any procedural programming or scripting language with Unity REST API for automation, management, based on organization requirement without owning market available tools.

Initially PowerShell is an automation platform and scripting language for Windows and Windows Server. Now PowerShell became a cross-platform (Windows, Linux, and MacOS) automation and configuration tool/framework that works well with all our existing tools and is optimized for dealing with structured data (e.g. JSON, CSV, XML, etc.), REST APIs, and object models. It includes a command-line shell, an associated scripting language and a framework for processing cmdlets, pronounced as “command-let”.

PowerShell Modules

A module is a set of related Windows PowerShell functionalities, grouped together as a convenient unit (usually saved in a single directory). By defining a set of related script files, assemblies, and related resources as a module, you can reference, load, persist, and share your code much easier than you would otherwise.

The main purpose of a module is to allow the modularization (i.e. reuse and abstraction) of Windows PowerShell code. For example, the most basic way of creating a module is to simply save a Windows PowerShell script as a .psm1 file. Doing so allows you to control (i.e. make public or private) the functions and variables contained in the script. Saving the script as a .psm1 file also allows you to control the scope of certain variables. Finally, you can also use cmdlets such as Install-Module to organize, install, and use your script as building blocks for larger solutions.

Generally *Invoke-RestMethod* and *Invoke-WebRequest* cmdlets will be used in PowerShell as a REST client to control over web requests. For Unity REST API, these cmdlets will not provide enough control so we have to create our own objects with .Net framework.

Integration of PowerShell Modules with Unity

Prerequisites

To integrate PowerShell modules with Unity REST API, we have very basic requirements such as,

- PowerShell version 5 or more than that

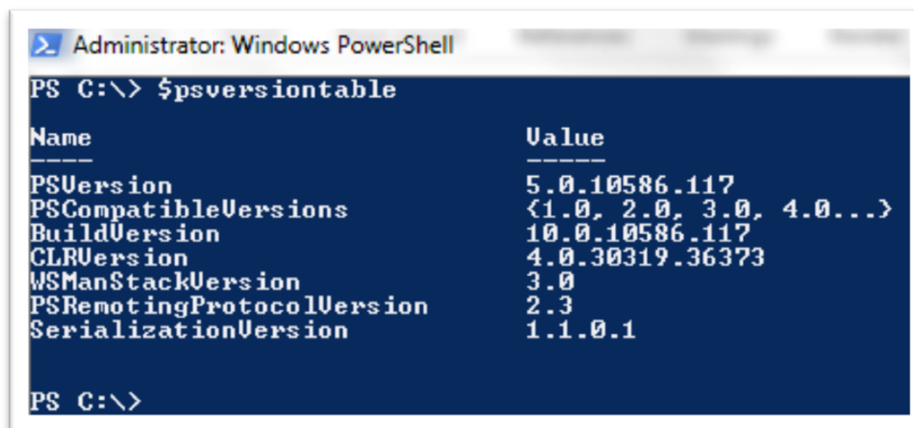


Figure 3: PowerShell version

- Dell EMC Unity array (Virtual or Physical array)

How to install PowerShell Modules

In PowerShell framework, installation referred as installing the modules which is the group of functionalities we develop. We have automatic and manual installation options for installing modules. In automatic installation, we use the location of repository, environmental path, module path and so on in the root module. In manual installation, we have to do those things manually.

```
#Automated installation (Powershell 5):
    Install-Module Unity-Powershell

# Or manual setup
    # Download the repository
    # Unblock the zip
    # Extract the Unity-Powershell folder to a module path (e.g. $env:USERPROFILE\Documents\WindowsPowerShell\Modules\

# Import the module
    Import-Module Unity-Powershell  #Alternatively, Import-Module \\Path\To\Unity-Powershell

# Get commands in the module
    Get-Command -Module Unity-Powershell

# Get help
    Get-Help Get-UnityUser -Full
    Get-Help Unity-Powershell
```

Figure 4: Installing the PowerShell Modules

We can check the installed modules in our PC/Desktop/Server as below,

```
PS C:\> Get-Module -ListAvailable

    Directory: C:\Program Files\WindowsPowerShell\Modules

ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.0.26     BuildHelpers                        {Export-Metadata, Get-BuildVariables, Get-GitChangedFile, ..
Script     1.1.1.0    PackageManagement                   {Find-Package, Get-Package, Get-PackageProvider, Get-Packa..
Binary     1.0.0.1    PackageManagement                   {Find-Package, Get-Package, Get-PackageProvider, Get-Packa..
Script     3.4.3      Pester                              {Describe, Context, It, Should...}
Script     1.1.2.0    PowerShellGet                       {Install-Module, Find-Module, Save-Module, Update-Module...}
Script     1.0.0.1    PowerShellGet                       {Install-Module, Find-Module, Save-Module, Update-Module...}
Script     4.6.0      psake                               {Invoke-psake, Invoke-Task, Get-PSakeScriptTasks, Task...}
Script     0.1.24     PSDeploy                            {By, DependingOn, Deploy, FromSource...}
Script     0.12.0     Unity-Powershell                    {Connect-Unity, Disable-UnityFastCache, Disconnect-Unity, ..
Script     0.11.1     Unity-Powershell                    {Connect-Unity, Disable-UnityFastCache, Disconnect-Unity, ..
```

Figure 5: List of PowerShell Modules

Working with Unity from PowerShell

Below are the sample scripts to connect Unity array which helps to understand how to declare variable and call the functions,

```
$Public  = @( Get-ChildItem -Path $PSScriptRoot\Public\*.ps1 -ErrorAction SilentlyContinue )
$Private = @( Get-ChildItem -Path $PSScriptRoot\Private\*.ps1 -ErrorAction SilentlyContinue )

Foreach($import in @($Public + $Private))
{
  Try
  {
    Write-Verbose "Import file: $($import.fullname)"
    . $import.fullname
  }
```

```
    Catch
    {
        Write-Error -Message "Failed to import file $($import.fullname): $_"
    }
}

Export-ModuleMember -Function $Public.Basename

[UnitySession[]]$global:DefaultUnitySession = @()

Class UnitySession

{
  [bool]$IsConnected

  [string]$Server

  [System.Collections.Hashtable]$Headers

  [System.Net.CookieCollection]$Cookies

  [Microsoft.PowerShell.Commands.WebRequestSession]$Websession

  [string]$SessionId

  [string]$User

  [string]$Name

  [string]$model

  [string]$SerialNumber

  [bool] TestConnection () {

    $URI = 'https://'+$This.Server+'/api/types/system/instances'

    Try {
        Invoke-WebRequest -Uri $URI -ContentType "application/json" -Websession $this.Websession -
Headers $this.Headers -Method 'GET'

    }

    Catch {

        $this.IsConnected = $false

        Write-Warning -Message "You are no longer connected to EMC Unity array: $($this.Server)"

        return $false
```

```
  }

  return $True
 }

}
```

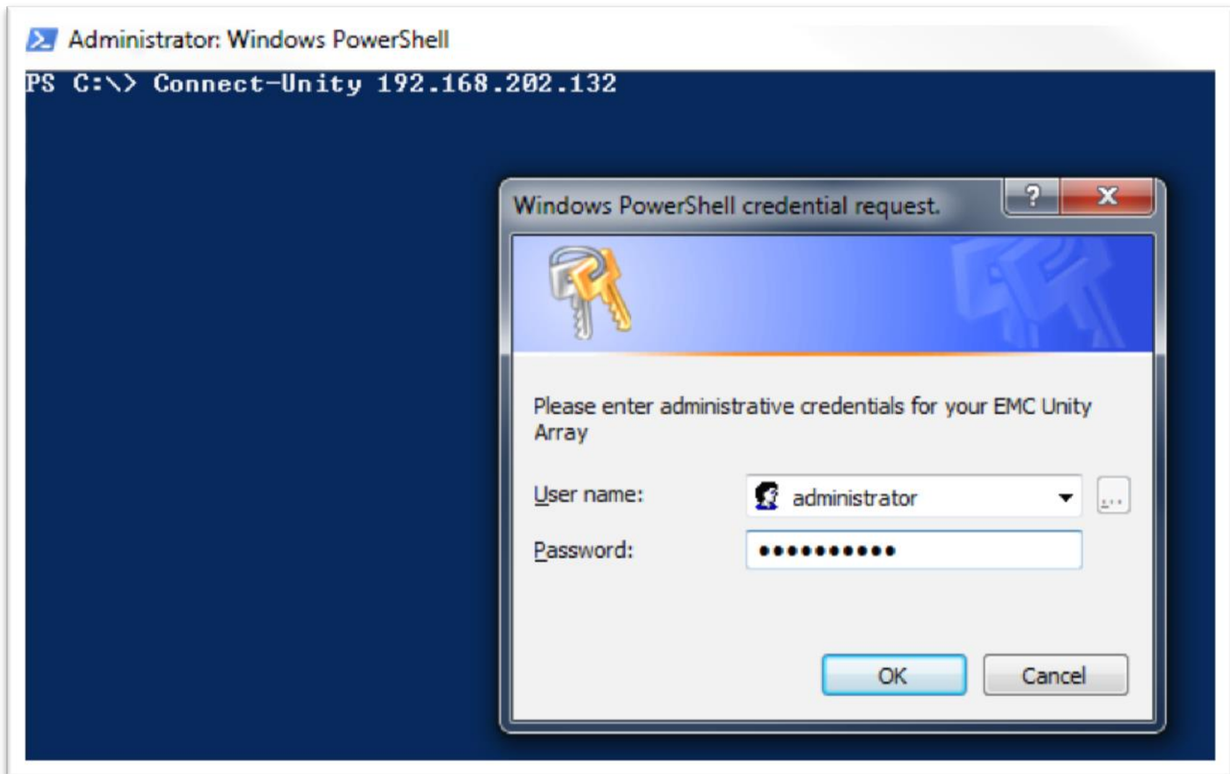


Figure 6: Unity Connectivity

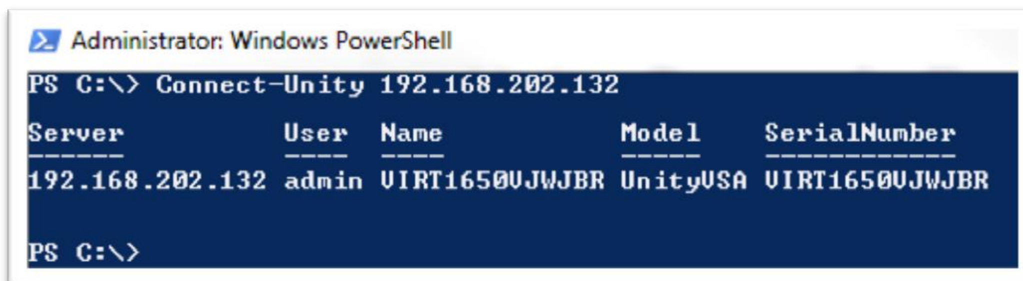


Figure 7: List of Unity arrays connected

Unity Administration Day 2 Task

In this section we will see a few sample PowerShell scripts like creating new NAS Server, new CIFS server. To run these scripts, you have to place this function in the root module and then functions will be imported while installing the modules.

Creating NAS Server

Below is the sample PowerShell script that use the REST API to successfully create NAS server on Unity array.

##To Create NAS Server

```
Function New-UnityNASServer {
[CmdletBinding(SupportsShouldProcess = $True,ConfirmImpact = 'High')]
  Param (
#Default Parameters
    [Parameter(Mandatory = $false,HelpMessage = 'EMC Unity Session')]
    $session = ($global:DefaultUnitySession | where-object {$_.IsConnected -eq $true}),
    [Parameter(Mandatory = $true,Position = 
1,ValueFromPipeline=$True,ValueFromPipelinebyPropertyName=$True,HelpMessage = 'Name for the 
NAS server')]
    [String[]]$Name,
    [Parameter(Mandatory = $true,HelpMessage = 'Storage processor ID on which the NAS server will 
run')]
    $homeSP,
    [Parameter(Mandatory = $true,HelpMessage = 'A Storage pool ID that stores NAS server 
configuration information')]
    [String]$Pool,
    [Parameter(Mandatory = $false,HelpMessage = 'Indicates whether the NAS server is a replication 
destination')]
    [bool]$isReplicationDestination,
    [Parameter(Mandatory = $false,HelpMessage = 'Directory Service used for quering identity 
information for Unix')]
    [NasServerUnixDirectoryServiceEnum]$UnixDirectoryService,
    [Parameter(Mandatory = $false,HelpMessage = 'Indicates whether multiprotocol sharing mode is 
enabled')]
    [bool]$isMultiProtocolEnabled,
    [Parameter(Mandatory = $false,HelpMessage = 'Use this flag to mandatorily disable access in case of 
any user mapping failure')]
    [bool]$allowUnmappedUser,
    [Parameter(Mandatory = $false,HelpMessage = 'Default Unix user name used for granting access in 
case of Windows to Unix user mapping failure')]
    [String]$defaultUnixUser,
    [Parameter(Mandatory = $false,HelpMessage = 'Default Windows user name used for granting 
access in case of Unix to Windows user mapping failure. When empty, access in such case is denied')]
    [String]$defaultWindowsUser
  )
  Begin {
    Write-Verbose "Executing function: $($MyInvocation.MyCommand)"
```

```
## Variables
   $URI = '/api/types/nasServer/instances'
   $Type = 'NAS Server'
   $StatusCode = 201
 }

 Process {
   Foreach ($sess in $session) {

     Write-Verbose "Processing Session: $($sess.Server) with SessionId: $($sess.SessionId)"

     Foreach ($n in $Name) {

#### REQUEST BODY

# Creation of the body hash
      $body = @{}

# Name parameter
      $body["name"] = "$($n)"

# homeSP parameter
      $body["homeSP"] = @{}
      $homeSPParameters = @{}
      $homeSPParameters["id"] = "$($homeSP)"
      $body["homeSP"] = $homeSPParameters

# Pool parameter
      $body["pool"] = @{}
      $poolParameters = @{}
      $poolParameters["id"] = "$($Pool)"
      $body["pool"] = $poolParameters

      If ($PSBoundParameters.ContainsKey('isReplicationDestination')) {
           $body["isReplicationDestination"] = $isReplicationDestination
      }

      If ($PSBoundParameters.ContainsKey('UnixDirectoryService')) {
           $body["currentUnixDirectoryService"] = $($UnixDirectoryService)
      }

      If ($PSBoundParameters.ContainsKey('isMultiProtocolEnabled')) {
           $body["isMultiProtocolEnabled"] = $isMultiProtocolEnabled
      }

      If ($PSBoundParameters.ContainsKey('allowUnmappedUser')) {
           $body["allowUnmappedUser"] = $allowUnmappedUser
      }

      If ($PSBoundParameters.ContainsKey('defaultUnixUser')) {
```

```
        $body["defaultUnixUser"] = $defaultUnixUser
    }

    If ($PSBoundParameters.ContainsKey('defaultWindowsUser')) {
        $body["defaultWindowsUser"] = $defaultWindowsUser
    }

#Show $body in verbose message
    $Json = $body | ConvertTo-Json -Depth 10
    Write-Verbose $Json

    If ($Sess.TestConnection()) {

##Building the URL
      $URL = 'https://'+$sess.Server+$URI
      Write-Verbose "URL: $URL"

#Sending the request
      If ($pscmdlet.ShouldProcess($Sess.Name,"Create $Type $n")) {
        $request = Send-UnityRequest -uri $URL -Session $Sess -Method 'POST' -Body $Body
      }
      Write-Verbose "Request status code: $($request.StatusCode)"
      If ($request.StatusCode -eq $StatusCode) {
#Formatting the result. Converting it from JSON to a Powershell object
        $results = ($request.content | ConvertFrom-Json).content

        Write-Verbose "$Type with the ID $($results.id) has been created"

        Get-UnityNASServer -Session $Sess -ID $results.id
      }
    }
   }
  }
 }
}
```

Output

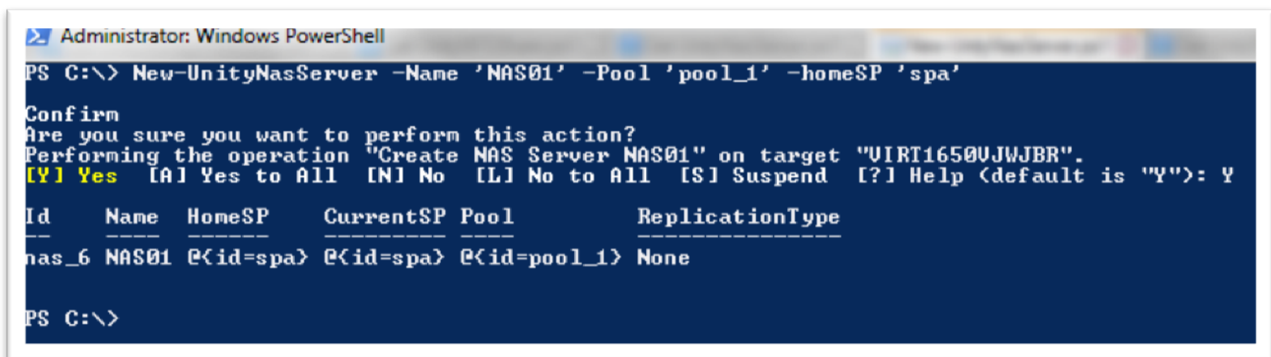


Figure 8: NAS Server creation with PowerShell Module

Creating CIFS Server

Below is the sample PowerShell script that uses the REST API to successfully create CIFS server on Unity array.

##To Create CIFS Server

```
Function Get-UnityCIFSShare {
[CmdletBinding(SupportsShouldProcess = $True,ConfirmImpact =
'High',DefaultParameterSetName="AD")]
Param (
#Default Parameters
   [Parameter(Mandatory = $false,HelpMessage = 'EMC Unity Session')]
   $session = ($global:DefaultUnitySession | where-object {$_.IsConnected -eq $true}),
   [Parameter(Mandatory = $false,Position =
1,ValueFromPipeline=$True,ValueFromPipelinebyPropertyName=$True,HelpMessage = 'User friendly,
descriptive name of SMB server')]
   [String[]]$Name,
   [Parameter(Mandatory = $true,HelpMessage = 'ID of the NAS server to which the SMB server
belongs')]
   [String]$nasServer,
   [Parameter(Mandatory = $false,HelpMessage = 'Computer name of the SMB server in Windows
network')]
   [String]$netbiosName,
   [Parameter(Mandatory = $false,HelpMessage = 'Description of the SMB server')]
   [String]$Description,
   [Parameter(Mandatory = $false,ParameterSetName="AD",HelpMessage = 'Domain name where
SMB server is registered in Active Directory, if applicable.')]
   [String]$domain,
   [Parameter(Mandatory = $false,ParameterSetName="AD",HelpMessage = 'LDAP organizational unit
of SMB server in Active Directory, if applicable')]
   [String]$organizationalUnit,
   [Parameter(Mandatory = $false,ParameterSetName="AD",HelpMessage = 'Active Directory domain
user name')]
   [String]$domainUsername,
```

```
   [Parameter(Mandatory = $false,ParameterSetName="AD",HelpMessage = 'Active Directory domain
password')]
   [String]$domainPassword,
   [Parameter(Mandatory = $false,ParameterSetName="AD",HelpMessage = 'Reuse existing SMB
server account in the Active Directory')]
   [Bool]$reuseComputerAccount,
   [Parameter(Mandatory = $false,ParameterSetName="Workgroup",HelpMessage = 'Standalone SMB
server workgroup name')]
   [String]$workgroup,
   [Parameter(Mandatory = $false,ParameterSetName="Workgroup",HelpMessage = 'Is Snapshot
Harvest Enabled')]
   [String]$localAdminPassword,
   [Parameter(Mandatory = $false,HelpMessage = 'List of file IP interfaces that service CIFS protocol of
SMB server')]
   [String[]]$interfaces
  )
Begin {
   Write-Verbose "Executing function: $($MyInvocation.MyCommand)"

## Variables
   $URI = '/api/types/cifsServer/instances'
   $Type = 'Server CIFS'
   $StatusCode = 201
  }

  Process {
   Foreach ($sess in $session) {

     Write-Verbose "Processing Session: $($sess.Server) with SessionId: $($sess.SessionId)"

     Foreach ($n in $Name) {

# Creation of the body hash
       $body = @{}

# nasServer argument
       $body["nasServer"] = @{}
       $nasServerArg = @{}
       $nasServerArg["id"] = "$($nasServer)"
       $body["nasServer"] = $nasServerArg

# netbiosName argument
       If ($PSBoundParameters.ContainsKey('netbiosName')) {
           $body["netbiosName"] = "$($netbiosName)"
       }

# Name argument
       If ($PSBoundParameters.ContainsKey('Name')) {
           $body["name"] = "$($name)"
       }
```

```
# Description argument
    If ($PSBoundParameters.ContainsKey('description')) {
        $body["description"] = "$($description)"
    }

# Domain argument
    If ($PSBoundParameters.ContainsKey('domain')) {
        $body["domain"] = "$($domain)"
    }

# Organizational Unit argument
    If ($PSBoundParameters.ContainsKey('organizationalUnit')) {
        $body["organizationalUnit"] = "$($organizationalUnit)"
    }

# Domain Username argument
    If ($PSBoundParameters.ContainsKey('domainUsername')) {
        $body["domainUsername"] = "$($domainUsername)"
    }

# Domain Password argument
    If ($PSBoundParameters.ContainsKey('domainPassword')) {
        $body["domainPassword"] = "$($domainPassword)"
    }

# Reuse Computer Account argument
    If ($PSBoundParameters.ContainsKey('reuseComputerAccount')) {
        $body["reuseComputerAccount"] = $reuseComputerAccount
    }

# Workgroup argument
    If ($PSBoundParameters.ContainsKey('workgroup')) {
        $body["workgroup"] = "$($workgroup)"
    }

# Local Admin Password argument
    If ($PSBoundParameters.ContainsKey('localAdminPassword')) {
        $body["localAdminPassword"] = "$($localAdminPassword)"
    }

#Interfaces argument
    If ($PSBoundParameters.ContainsKey('interfaces')) {
      $body['interfaces'] = @()
      Foreach ($int in $interfaces) {
        $ÃŽntArgument = @{}
        $ÃŽntArgument['id'] = "$($int)"
        $body["interfaces"] += $ÃŽntArgument
      }
    }
```

```
#Show $body in verbose message
      $Json = $body | ConvertTo-Json -Depth 10
      Write-Verbose $Json

      If ($Sess.TestConnection()) {

##Building the URL
        $URL = 'https://'+$sess.Server+$URI
        Write-Verbose "URL: $URL"

#Sending the request
        If ($pscmdlet.ShouldProcess($Sess.Name,"Create $Type $n")) {
          $request = Send-UnityRequest -uri $URL -Session $Sess -Method 'POST' -Body $Body
        }

        Write-Verbose "Request status code: $($request.StatusCode)"

        If ($request.StatusCode -eq $StatusCode) {

#Formating the result. Converting it from JSON to a Powershell object
          $results = ($request.content | ConvertFrom-Json).content

          Write-Verbose "$Type with the ID $($results.id) has been created"

          Get-UnityCifsServer -Session $Sess -ID $results.id
        }
      }
    }
  }
 }
}
```

Output

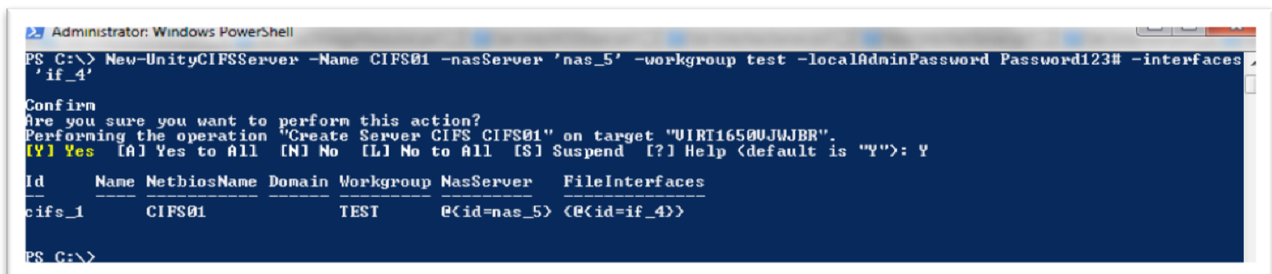


Figure 9: CIFS Server Creation with PowerShell Module

Summary

Through this document we have delivered the automation framework for Unity family (All Flash, Hybrid & VSA). Any IT infrastructure admin can use this as a reference document to come up with adding other day-to-day activities. This automation framework shall facilitate all the post configuration activities of the array. After this, admin shall utilize Dell EMCs SDS solution, i.e. VIPR to provision storage resources and services via catalog. Added to the above benefits, this article will help you understand and develop your own code to automate the Unity arrays. Provided sample codes and demos will drive through the play with codes to automate minimal complex day-to-day operations on Unity system.

Bibliography

- https://github.com/equelin/Unity-Powershell
- https://community.emc.com/docs/DOC-51784
- https://community.emc.com/docs/DOC-52469
- http://muegge.com/blog/emc-unity-rest-api-powershell
- http://ramblingcookiemonster.github.io/Building-A-PowerShell-Module/
- http://wahlnetwork.com/category/deep-dives/the-power-of-powershell/
- https://github.com/PowerShell/PowerShell/tree/master/docs/learning-powershell
- https://www.emc.com/collateral/white-papers/h15084-emc-unity-introduction-to-the-unity-platform.pdf

Dell EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED “AS IS.” DELL EMC MAKES NO RESPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying and distribution of any Dell EMC software described in this publication requires an applicable software license.

Dell, EMC and other trademarks are trademarks of Dell Inc. or its subsidiaries.