PowerCLI

THE ASPIRING AUTOMATOR'S GUIDE

SECOND EDITION
BY XAVIER AVRILLIER

DOJO VMware

ALTARO
Part of **HORNETSECURITY** group

INTRODUCTION

Automation, scripting and PowerCLI are words that most people working with VMware products know pretty well and have used once or twice. Everyone also knows automation is a great tool to have in your tool belt. The problem usually is that getting into automation and scripting appears daunting to many people who feel like the learning curve is just too steep and they usually don't know where to start. The good thing is you don't need to learn everything straight away to start working with PowerShell and PowerCLI. Once you have the basics down and have your curiosity tickled, you will learn what you need as you go... a lot faster than you thought you would!

This eBook doesn't contain absolutely everything about automation, but it serves as an ideal starting point explaining what need to know to get started. Just like a mechanic's starter kit, you can't build a car engine with it but you can do the basics to call yourself a budding mechanic. This book contains the first few steps of a long stairway paved with cmdlets, properties and methods and a little more than that.

SECOND EDITION UPDATE NOTES

PowerCLI: An Aspiring Automator's Guide was first released in 2019 and received widespread acclaim from the community. The aim of the original release was to deliver solid foundation skills for anyone to get started with PowerCLI and PowerShell. In this second edition, a full revision of the original text has been undertaken as well as updated app features and additional content on actions that go a bit more in-depth compared to just how to use the standard built-in cmdlets.

Here is an overview of the additional content you can expect to find in this edition:

- What's new in PowerCLI 12.2
- Working with vCenter HA (VCHA)
- Reporting on Raw Device Mapping (RDM)
- Backup vSphere host configuration
- Reporting on snapshots
- Getting datastore usage information
- Container based vCenter simulator (VCSIM)

CONTENTS

INTRODUCTION **3**

SECOND EDITION UPDATE NOTES **4**

CHAPTER 1: INTRODUCTION TO POWERCLI **9**

What is PowerCLI? **9**

Why would you want to use it? **10**

What can you do with it? **11**

PowerCLI 12.2 ..... **11**

CHAPTER 2: INSTALLING POWERCLI **14**

Requirements **14**

Before Installing **14**

Downloading and Installation **14**

Machine with Internet Access 15

Machine with no Internet access 15

Execution policy 16

CHAPTER 3: GETTING STARTED WITH POWERCLI **18**

Configuring PowerCLI **18**

Connecting to vCenter **20**

Credential store **21**

PowerCLI Objects **21**

Object-By-Name selection (OBN) 21

Data structure 22

Object State 23

Retrieving Basic Data on vSphere Objects **24**

List All Powered-on Virtual Machines with their Folders in Cluster "Management" 25

List Virtual Machines and the Hosts they are Running on 25

Exploring PowerCLI Objects 26

Retrieve Advanced Data on vSphere Objects (ExtensionData) **30**

CHAPTER 4: WRITING SCRIPTS **33**

Shorter is better **33**

No Hardcoded Values **33**

Error Handling **34**

PSScriptAnalyzer **34**

Functions **34**

Why use Functions? 35

Functions Best Practices 36

CHAPTER 5: HOW TO USE SCHEDULED TASKS AND HTML REPORTS **40**

Task Scheduling **40**

Preparation 40

Installing the scripts 42

Create the scheduled task 43

HTML reporting 45

Producing the HTML report 46

Processing the report 46

CHAPTER 6: USE CASES 49

Deploying virtual machines from a CSV file 49

Requirements 49

The Script Explained 50

Manage vSphere tags 54

Add a portgroup on a standard vSwitch on all hosts in a cluster 57

Monitor VM CPU usage in near real time 59

Retrieve stats 59

Progress bar 62

Monitor multiple counters 63

Change the VSAN default rebuild delay 64

vCenter High Availability (VCHA) 66

The properties and methods 67

Get vCenter HA status information 68

Retrieve vCenter HA node information 70

Going further 71

Reporting on Raw Device Mapping disks 73

Identifying VMs with an RDM disk 74

Obtaining information about the RDMs on a VM 74

Backup vSphere Host Configuration **76**

Backing up a vSphere host configuration to file 76

Wrapping it into a scheduled task 77

Restoring a vSphere host configuration 79

Reporting on snapshots **79**

Retrieving usage information of a datastore **82**

CHAPTER 7: RUNNING ESXCLI COMMANDS IN POWERCLI **85**

Browsing ESXCLI **85**

Running commands **86**

Help() 86

Invoke() 87

CreateArgs() 88

vCenter and ESXi API based simulator **90**

Docker engine 91

Deploy the VCSIM container 92

Using PowerCLI with VCSIM 93

CHAPTER 8: CONTINUING YOUR AUTOMATION JOURNEY **96**

ABOUT XAVIER AVRILLIER **98**

ABOUT ALTARO **101**

CHAPTER 1: INTRODUCTION TO POWERCLI

Let's get to know PowerCLI a little better before we start getting our hands dirty in the command prompt. If you are reading this, you probably already know what PowerCLI is about or have a vague idea of it but if not, we'll quickly get you up to speed. After a while working with PowerCLI, it becomes second nature and you can't really imagine life without it anymore. You then forget that many IT folks out there don't use PowerCLI or simply haven't heard of it. Thanks to VMware's drive to push automation, the product's integration with all of their components grew bigger and better over the years and it has become a critical part of their ecosystem.

WHAT IS POWERCLI?

As opposed to what some may think, PowerCLI is not a software but a command-line and scripting tool built on Windows PowerShell for managing and automating vSphere environments. It used to be distributed as an executable file to install on the workstation. An icon was generated that would essentially launch PowerShell and load the PowerCLI snap-ins in the session. This behaviour changed back in version 6.5.1 when the executable file was removed and replaced by a suite of PowerShell modules to install from within the prompt since they were added to the PowerShell Gallery (Microsoft's registered repository).

These modules provide the means to interact with the components of a VMware environment and offer more than 600 cmdlets!

```
PS> Get-Command -Module (Get-Module vmware* -ListAvailable) | Measure-Object

Count    : 891
```

WHY WOULD YOU WANT TO USE IT?

Apart from the fact that it has become an industry standard and is a welcome addition to anyone's resume, there are many compelling reasons why one should learn PowerShell, and PowerCLI by extension. Let's focus on its three core benefits:

SAVE TIME ON REPETITIVE TASKS

Virtualization is generally associated with cool words like system architecture, design and troubleshooting of complex environments. While this is definitely true, the operational side of it is often forgotten because it is a lot less attractive and consists mostly of monotonous and time-consuming tasks. Unfortunately, it has to be done and in a vast majority of cases the vSphere administrator will wear both the project work hat and operational work hat. These repetitive operations include boring tasks like datastore and snapshots monitoring, VM creation, host implementation, you name it. All these little things that if not done could quickly ruin your day.

PowerCLI will allow you to automate most of them, be it in the command prompt or in a script that can be run in a schedule task. People are often surprised when they see the time they could save and use it to do more interesting things (like writing more scripts to automate even more!).

AVOID HUMAN ERROR

When something goes wrong, 9 times out of 10 it is due to human error. Obviously, you will never be free from the possibility of a hardware component failing or a bug in a firmware, but these are events you cannot prevent. What PowerCLI can help you with is consistency and avoiding silly mistake. What if you detached the wrong LUN in the vSphere client because you read "b" instead of "d" and 15 VMs go down… This wouldn't have happened if you gathered the Lon ID from a datastore object in PowerCLI. For example, what if your colleague configured a standard port group with VLAN ID 148 instead of 1048

and the VM isn't recovered in an HA event because of this? A simple script would ensure all port-groups are created the same way from the start.

IMPROVE REPORTING AND ALERTING

Every company running a vSphere infrastructure should have a solid agent-based monitoring system like Zabbix or Nagios and enable vCenter alerts at the very least. Complementary solutions like vRealize Operations Manager can greatly improve your visibility of the environment, but as great as they are, these come at a cost and may not allow reporting on the information you are after.

Once again, PowerCLI can increase your reporting and alerting capabilities to fill the gaps in your monitoring. You will be able to create well formatted HTML reports served on a web page or sent via email.

WHAT CAN YOU DO WITH IT?

In PowerCLI you can do just about anything you can do in the web client and more. You can generally get the information you are after or change a configuration much quicker than in the vSphere client. The most common example is automating the deployment of a large number of VMs which we will talk about more extensively in the use cases.

POWERCLI 12.2

The latest version of PowerCLI (at the time of publication) is PowerCLI 12.2. This version brings a number of new features as well as support for newly released products and new product versions:

Added cmdlets for:

- ESXi host network management
- HCX management
- Namespace Management
- Trusted Host Services management
- VM Guest Disk management
- VMware Cloud on AWS management
- vSAN management
- DRaaS management
- vSphere Lifecycle Manager (vLCM)
- Workload Management clusters management

Updated support for:

- vSphere 7.0
- DRaaS
- VMware Horizon PowerCLI module now supported for macOS and Linux
- vRealize Operations 8.0
- License module for multiplatform use
- vROps module for multiplatform use
- Open-VMConsoleWindow for multiplatform use
- Move-VM to include opaque networks
- PowerShell versions

This version also includes a new VMware Cloud Services Management module that contains 11 new cmdlets. By default, it will connect to the public “console.cloud.vmware.com” if no server is specified in the “Connect-Vcs” cmdlet.

```
PS> Get-Command -Module VMware.CloudServices

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Connect-Vcs                                        12.1.0.... VMware.CloudServices
Cmdlet          Disconnect-Vcs                                     12.1.0.... VMware.CloudServices
Cmdlet          Get-VcsOrganizationRole                            12.1.0.... VMware.CloudServices
Cmdlet          Get-VcsService                                     12.1.0.... VMware.CloudServices
Cmdlet          Get-VcsServiceRole                                 12.1.0.... VMware.CloudServices
Cmdlet          Get-VcsUser                                        12.1.0.... VMware.CloudServices
Cmdlet          Get-VcsUserInvitation                              12.1.0.... VMware.CloudServices
Cmdlet          New-VcsOAuthSecurityContext                        12.1.0.... VMware.CloudServices
Cmdlet          New-VcsUserInvitation                              12.1.0.... VMware.CloudServices
Cmdlet          Remove-VcsUser                                     12.1.0.... VMware.CloudServices
Cmdlet          Remove-VcsUserInvitation                           12.1.0.... VMware.CloudServices
```

If you are using scripts in production that were written with PowerCLI 11.x or earlier, as usual we recommend that you always check that they work correctly with the new version before updating PowerCLI on those systems. It is best to discover a compatibility problem interactively than a few weeks down the line where you realize a script hasn't been running properly.

You can find the procedure on how to update PowerCLI in the Downloading and Installation section of this book.

CHAPTER 2: INSTALLING POWERCLI

REQUIREMENTS

Before you begin, find all the requirements and interoperability information about PowerCLI in the compatibility matrix for PowerCLI to make sure your configuration is supported. Head over to VMware code, select your version of PowerCLI and open the associated compatibility matrix.

You will need .NET Framework 4.7.2 and above. Since version 11.2.0, PowerCLI is officially supported on PowerShell version 4.0 and above as well as Windows 10 / Server 2012 R2 and above.

Note that PowerCLI is now also available on Linux and macOS thanks to PowerShell Core.

BEFORE INSTALLING

Prior to installing the latest version, you need to uninstall any version of PowerCLI older than 6.5 R1 from your system. You can uninstall it like any installed software in "Programs and Features".

DOWNLOADING AND INSTALLATION

The installation procedure will be slightly different with regards to whether the machine you are installing it on has internet access or not. Let's cover both scenarios.

MACHINE WITH INTERNET ACCESS

INSTALLING

The installation procedure has been simplified since the modules have been added to the PowerShell Gallery with version 6.5.1 and is now straightforward.

- Open a PowerShell prompt and install the modules using:

```
Install-Module VMware.PowerCLI -Scope CurrentUser
```

The modules will be automatically downloaded and stored in the correct folder.

> Note that you can use the -Scope parameter to make the PowerCLI modules available to AllUsers.

UPDATING

Although it will work, it is recommended to avoid using the Update-Module cmdlet to update PowerCLI as it will not remove any files rendered obsolete by the new version. Therefore,

- Uninstall the existing version using:

```
Get-module VMware.* -listAvailable | Uninstall-Module -Force
```

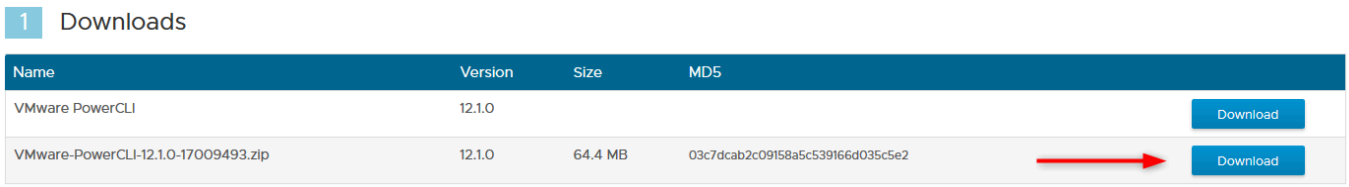
- Next, install the new version by following the install procedure outlined previously

MACHINE WITH NO INTERNET ACCESS

INSTALLING

If your system does not have Internet access you will need to download PowerCLI as a zip file from the VMware website and copy the modules into your modules folder. Unlike many of the VMware products, you don't need to be logged in to download PowerCLI.

- Head over to VMware code and select the latest version of PowerCLI
- Download the zip file



- Transfer the file to your offline machine and copy all the folders present in the zip to your PowerShell Modules folder. Again, choose the location accordingly to make it available to everyone or to yourself only:

Current User	%USERPROFILE%\Documents\WindowsPowerShell\Modules
All Users	C:\Program Files\WindowsPowerShell\Modules

UPDATING

To update PowerCLI, delete the existing PowerCLI module folders and follow the procedure outlined above.

EXECUTION POLICY

Execution policies are a security mechanism that determines if files can be loaded in PowerShell such as config files, modules, scripts… You can find your current execution policy by running:

```
Get-ExecutionPolicy
```

You may need to change the default execution policy to be able run scripts you wrote. Unless a GPO changed the default setting, you won't need to if you are on a Windows Server OS. However, it is required

for Windows client OS (i.e. Windows 10) as the default is set to Restricted, you will then need to change it to RemoteSigned with:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

Some highly secured environments may require that all the scripts (even those written in-house) are digitally signed by the enterprise PKI. This means the execution policy must be set to AllSigned if not already done via GPO. In which case you will have to digitally sign your scripts prior to running them.

> **Note:** Any change in the script means a new signature is required.

CHAPTER 3: GETTING STARTED WITH POWERCLI

CONFIGURING POWERCLI

Apart from the execution policy mentioned previously, there is no configuration required before using PowerCLI. However, it is worth talking about a few recommended settings that could be useful. You can use the *Set-PowerCLIConfiguration* cmdlet to amend the configuration of PowerCLI.

Once again, a "*-scope*" parameter will allow you to modify the settings for the *session*, *current user* or *all users*. The *Session* scope has the highest priority, and the *All Users* scope has the lowest of the three.

> **Note:** Any setting configured with the Session scope will not be persisted after closing the current prompt.

Other parameters allow you to configure settings such as whether or not to use the system proxy, display warnings about deprecated elements, increase the timeout of commands (which defaults at 5 minutes), etc. You can find all the parameters using:

```
Get-help Set-PowerCLIConfiguration -parameters *
```

Let's take a look at two of the most common parameters you'll use here.

```
-DefaultVIServerMode
```

This parameter is sometimes overlooked, but it is very important if your environment contains more than one vCenter server. **This parameter defines whether you can connect to more than one vCenter at a time.** It is important because a mistake can quickly occur if you have forgotten that you have two active vCenter sessions in your PowerShell prompt, for example. You typically won't have problems with Get commands where you're just retrieving information, but you REALLY need to start paying attention once you start working with Set, Remove…etc…etc in this situation.

Restrict to one vCenter	`Set-PowerCLIConfiguration -DefaultVIServerMode Single`
Allow multiple vCenter	`Set-PowerCLIConfiguration -DefaultVIServerMode Multiple`

`-InvalidCertificateAction`

This parameter defines the behaviour of the Connect-VIServer with regards to the vCenter certificate. This parameter does not apply to you if the installed vCenter certificate is signed by a trusted root CA.
If the certificate isn't valid (expired, self-signed, not trusted, etc.) the default setting is to display a warning when you establish the connection. You can change this behaviour with the following parameter:

Prompt before continuing. Possible choices are Deny, Accept for once, Accept permanently, Accept for all users.	`Set-PowerCLIConfiguration -InvalidCertificateAction Prompt`
No connection established	`Set-PowerCLIConfiguration -InvalidCertificateAction Fail`
Connection established regardless of the certificate status	`Set-PowerCLIConfiguration -InvalidCertificateAction Ignore`
(Default value) Connection is established but a warning is displayed	`Set-PowerCLIConfiguration -InvalidCertificateAction Warn`

CONNECTING TO VCENTER

You can connect to a ESXi host or a vCenter server with the *Connect-VIServer* cmdlet. You can do it a few different ways thanks to the various parameters it offers. You can connect with a Session ID, a credential object, a user password/combination, credential store, etc.

The most common way to connect is to specify the IP or FQDN of the server and use the current PowerShell session user if it has permissions on the vCenter server:

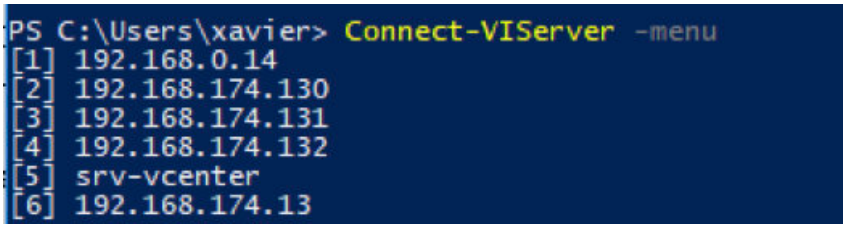
```
Connect-VIServer vcsa.test.local
```

You can retrieve the list of the connected vCenter servers by displaying the content of the global variable **`$Global:DefaultVIServers`**, which populates after a successful connection (a tiny caveat here: this is a variable). This means that if a network interruption occurs, for instance, and the session is no longer connected, the variable will remain populated, and in which case you will need to reconnect, causing the variable to be overwritten with the new active session. This object contains information about the server and the session such as the session ID, user, version and build… If it cannot connect with the current user, a credential popup appears to let you enter a different user/password combination. You can also add the credential parameter, so you don't have to wait for the current user to fail the connection:

```
Connect-VIServer vcsa.test.local –Credential (Get-Credential)
```

I would also like to point out a nifty little parameter of this cmdlet that displays a list of the recently connected vCenter servers. You don't see this one too often, but it can be very useful if you manage a bunch of different servers. You can then select the one you want to connect to with its number

```
Connect-VIServer -menu
```



CREDENTIAL STORE

By default, when connecting to a vCenter server, PowerCLI will check what is referred to as the Credential Store. Similar to the Windows Credentials Manager, the Credential Store saves sets of host credentials you have at your disposal. There are 2 main benefits to using the Credential Store:

1. Credentials are encrypted and stored securely (never put credentials in your scripts!)
2. Connect faster to a server by not having to enter credentials

Use the Credential Store if your domain user has no access permissions to vCenter.

To use the credential store, you will need to save credentials into it, as shown below

```
New-VICredentialStoreItem -Host 192.168.0.16 -User "John" -Password
"Don'tL00kAtM3!"
```

Once it is stored you can just connect to the vCenter with the usual command:

```
Connect-VIServer srv-vcenter.domain.pri
```

POWERCLI OBJECTS

Just like with traditional PowerShell, you will be constantly working with PowerShell objects. Any object acquired with a PowerCLI cmdlet will be assigned the specific type associated with their nature (virtual machine, datastore, GuestID…). You'll discover more of them as you go deeper into PowerCLI.

OBJECT-BY-NAME SELECTION (OBN)

PowerCLI includes a mechanism called OBN selection that allows you to refer to an object by its name instead of passing the actual object through the pipeline or as a parameter. It simplifies and makes

it easier to write scripts and commands in the prompt. OBN selection also supports wildcards and multiple arguments as shown here:

- **Traditional Selection**

```
Move-VM -VM (Get-VM -name "srv-vm-1","srv-vm-2") -Destination (Get-
VMHost -Name "ESX-Host-1")
```

- **OBN Selection**

```
Move-VM -VM "srv-vm-1","srv-vm-2" -Destination "ESX-Host-1"
```

DATA STRUCTURE

Data structure is not something you have to worry about when initially working with PowerCLI. However, it is important to be aware of the object types to understand what is going on behind the scenes. For that reason, we won't go too deep into the theory and summarize instead for the sake of brevity.

There are four category types for objects in PowerCLI, and each serves a different purpose:

1. **Managed Object Types** - Make managing objects possible. They can contain both properties and methods, meaning they can hold values (VM name, Data objects) or run an action (Start VM). A managed object is identified by an ID called "*Managed Object Reference*", generally referred to as *MoRef* which is a data object (see below).

2. **Data Object Types** - As the name suggests, they hold data about the object. They can contain the MoRef (which allows you to get a managed object) or other data objects (mac address, vmx file path, etc.).

3. **Enumerated Object Type** - Data Objects that can only store values or objects from a pre-defined set and nothing else. They are usually reserved for runtime statuses such as the power state of a virtual machine that can only be "powered on", "powered off" or "suspended".

4. **Fault Types** - Provides information about errors. It also contains a data object (property) with the MoRef of the respective managed object.

OBJECT STATE

Generally in PowerShell, it is very common to store objects in variables rather than running the cmdlet every time to retrieve it. It saves time as a variable name is usually shorter to enter than a cmdlet, and there is no processing time because it is already stored locally. When doing so, you are storing its state at this point in time. Meaning that if a property were to change on the server-side object, it would not be reflected on the object in the variable. You would then need to run the cmdlet again to "update" the content of the variable.

EXAMPLE WITH VMOTION

1. Retrieve a VM object in a variable:

```
PS C:\> $VM = Get-VM "test-vm"
```

2. Display the host it is currently running on:

```
PS C:\> $VM | select vmhost

VMHost
------
ESX-Host-1
```

3. Migrate the VM to another host:

```
PS C:\> $VM | Move-VM -Destination " ESX-Host-2"
```

4. Run the second command again:

(As you can see it is still showing the host it was previously running on)

```
PS C:\> $VM | select vmhost

VMHost
------
ESX-Host-1
```

5. Update the variable and display the host value once again:

(The variable now reflects the actual value of the server side object)

```
PS C:\> $VM = Get-VM "test-vm"

PS C:\> $VM | select vmhost

VMHost
------
ESX-Host-2
```

You can see how this could cause confusion on systems managed by several administrators, with reporting scripts showing outdated data about the environment, for example.

> **Note:** This is true for all kind of objects, not only virtual machines.

RETRIEVING BASIC DATA ON VSPHERE OBJECTS

As we get into this next section, a quick tip for beginners when starting off with PowerCLI -especially if you are fairly new to PowerShell as well. If you don't have a test lab to practice with, I recommend being on the safe side and sticking with *Get* commands for a little while. Build your confidence and get comfortable with how it works before moving on to cmdlets that change the state of objects like Set or Remove. A mistake could have consequences on operations.

This chapter only includes basic PowerCLI cmdlets to give you an idea of what you can do. I will also take the opportunity of using these simple commands to show you a few concepts applied to the prompt. To get a complete list of the PowerCLI cmdlets, type the following command (I recommend to use *Out-GridView* for better readability):

```
Get-Command -Module VMware* | Out-GridView
```

LIST ALL POWERED-ON VIRTUAL MACHINES WITH THEIR FOLDERS IN CLUSTER "MANAGEMENT"

The best place to start in PowerCLI is to list the VMs in the environment. This cmdlet is probably one of the most commonly used. It includes a lot of information about your servers. Similarly with traditional PowerShell commands, you can use the pipe " | " to narrow down the results to only what you are after, for example:

```
Get-Cluster "Management" | Get-VM | Where PowerState -eq "PoweredOn"
| Select Name,Folder
```

In this example we list the cluster named "*Management*", then we list the VMs registered in this cluster, then we filter only the ones that have the property "*Powerstate*" (Enumerated object) set to "*PoweredOn*". Virtual machines include lots of useful properties in their "base" object (extended properties excluded - more on that later).

LIST VIRTUAL MACHINES AND THE HOSTS THEY ARE RUNNING ON

Another simple cmdlet that you'll find useful display more about PowerCLI objects in context this time. As mentioned previously, virtual machine objects contain many interesting properties, including the host they are running on. In PowerCLI, hosts are referred to as *VMHost*.

```
PS C:\> $VM | select name,vmhost

Name     VMHost
----     ------
Test-vm  ESX-Host-2
```

The interesting thing with the *VMHost* property is that, even though the name of the host is displayed, it actually contains a *VMHost* object. Meaning you can obtain information about the host by working with the *VMHost* property - no need to retrieve the object with the *Get-VMHost* cmdlet.

In other words, the following two commands return the same result. In the first command, we retrieve the *VMHost* using the cmdlet on the *VM* object In the second, we display the *VMHost* property of the *VM* object.

```
$VM | Get-VMHost

$VM.VMHost
```

Of course, this is not only true for *VMHost* but for most object types like clusters, resource pools, virtual switches, etc.

> **Note:** When using variables you can't use **`$Host`** as it is already a PowerShell automatic variable. Use **`$VMHost`** instead.

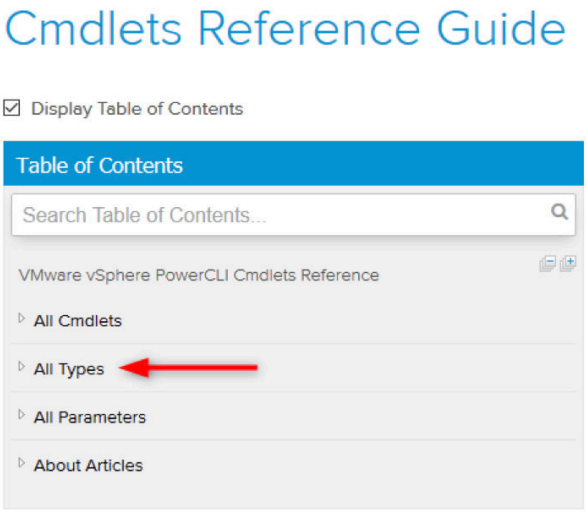
EXPLORING POWERCLI OBJECTS

Now that we touched base on a few basic cmdlets, it is time to start poking around in the PowerCLI objects so you can be able to find what you are after by yourself. There are essentially two ways of discovering PowerCLI object properties. You can either look it up in the online documentation or explore the object itself from within the prompt.

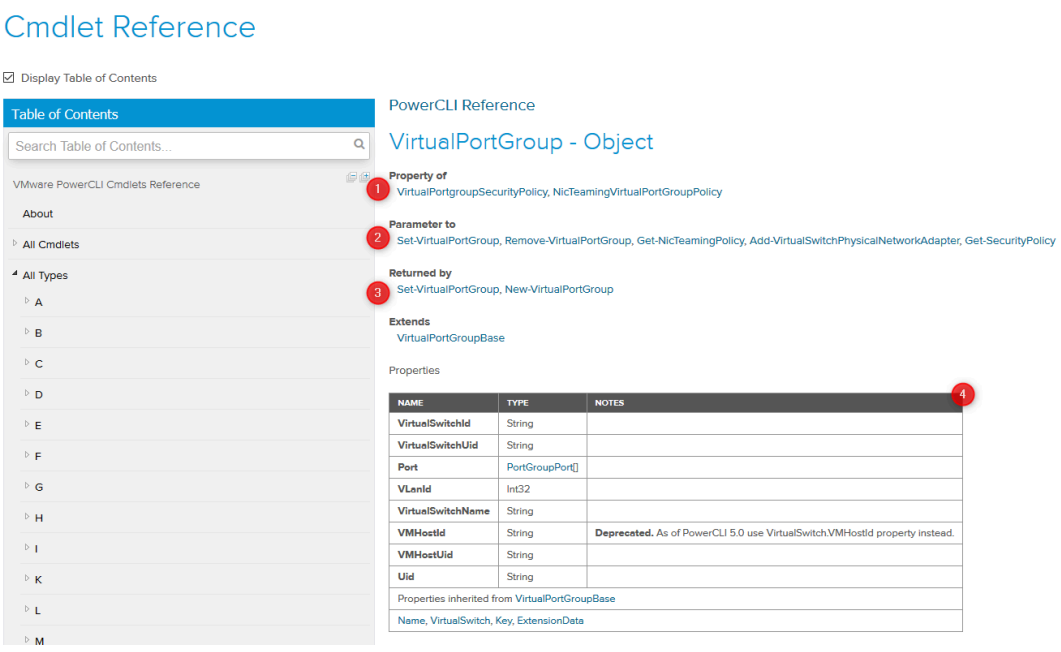
1. VMware cmdlets reference guide

VMware provides a reference guide listing all the cmdlets and properties in PowerCLI. It's very useful as it consolidates all the information about cmdlets, parameters and object types. For instance, if you want to find information about vSphere portgroups.

- Expand the "All *Types*" tab and find the "*VirtualPortGroup*" link. It is sorted alphabetically so pretty convenient.



- After clicking the *VirtualPortGroup* link, you'll get information about the type.



- Here we learn that:

  1) The Virtual Port Group Object be returned as a property of these PowerCLI objects (on which you can click for more information).

  2) It can be specified as a parameter of these cmdlets.

  3) This type of object is returned by the *Get-VirtualPortGroup* and *New-VirtualPortGroup* cmdlets. If you click on the cmdlet link you will be redirected to its documentation page.

  4) It shows the properties contained by the type of objects.

2. PowerShell Prompt

Now if you don't want to get out of your PowerShell prompt (and be a real automator), you can find the same information with some detective work.

- Start by looking for the cmdlet that will help you find the information using *Get-Command*. Then filter the cmdlets matching *Get*portgroup** (case insensitive) that are provided by the VMware modules.

```
PS C:\> Get-Command -Name Get*portgroup* -Module vmware.*
CommandType     Name                                               Ve
-----------     ----                                               --
Cmdlet          Get-VDPortgroup                                    12.1.0....
Cmdlet          Get-VDPortgroupOverridePolicy                      12.1.0....
Cmdlet          Get-VirtualPortGroup                               12.1.0....
Cmdlet          Get-VMHostProfileVmPortGroupConfiguration          12.1.0....
```

As you can see, several cmdlets match the filter, but a quick *Get-Help* on each of them will tell you that *Get-VirtualPortGroup* is the one you are after.

- Next, run the command on a portgroup and explore the object using *Get-Member.*

  The output will list the properties and methods of the object. You will see basic information on them such as object type for properties, object type returned by a method, arguments required or not by a method (more on that later).

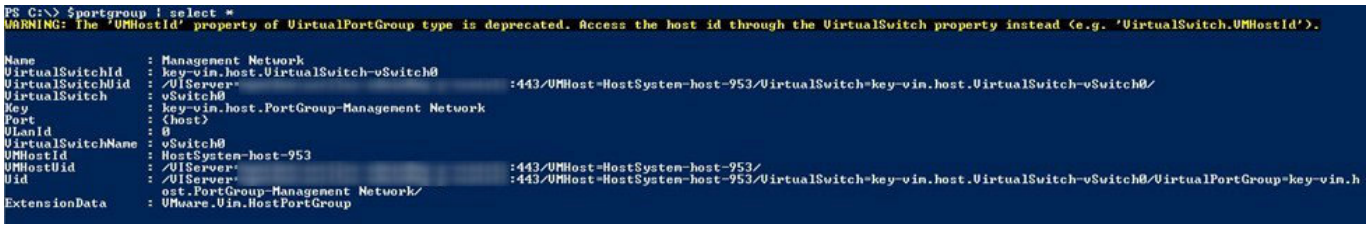
```
$PortGroup = Get-VirtualPortGroup -Name "Management Network" -VMHost
ESX-Host-1

$PortGroup | Get-Member
```

```
PS C:\> $portgroup | Get-Member

   TypeName: VMware.VimAutomation.ViCore.Impl.V1.Host.Networking.VirtualPortGroupImpl

Name                 MemberType Definition
----                 ---------- ----------
AttachNetworkAdapter Method     VMware.VimAutomation.ViCore.Interop.V1.Task.TaskInterop VirtualP
ConvertToVersion     Method     T ConvertToVersion[T](), T VersionedObjectInterop.ConvertToVersi
Equals               Method     bool Equals(System.Object obj)
GetClient            Method     VMware.VimAutomation.ViCore.Interop.V1.VIAutomation VIObjectCore
GetHashCode          Method     int GetHashCode()
GetType              Method     type GetType()
IsConvertableTo      Method     bool IsConvertableTo(type toType), bool VersionedObjectInterop.I
LockUpdates          Method     void ExtensionData.LockUpdates()
ToNetworkAdapter     Method     System.Collections.Generic.IEnumerable[VMware.VimAutomation.ViCo
ToString             Method     string ToString()
ToVirtualMachine     Method     System.Collections.Generic.IEnumerable[VMware.VimAutomation.ViCo
ToVirtualSwitch      Method     System.Collections.Generic.IEnumerable[VMware.VimAutomation.ViCo
UnlockUpdates        Method     void ExtensionData.UnlockUpdates()
ExtensionData        Property   System.Object ExtensionData {get;}
Key                  Property   string Key {get;}
Name                 Property   string Name {get;}
Port                 Property   VMware.VimAutomation.ViCore.Types.V1.Host.Networking.PortGroupPo
Uid                  Property   string Uid {get;}
VirtualSwitch        Property   VMware.VimAutomation.ViCore.Types.V1.Host.Networking.VirtualSwit
VirtualSwitchId      Property   string VirtualSwitchId {get;}
VirtualSwitchName    Property   string VirtualSwitchName {get;}
VirtualSwitchUid     Property   string VirtualSwitchUid {get;}
VLanId               Property   int VLanId {get;}
VMHostId             Property   string VMHostId {get;}
VMHostUid            Property   string VMHostUid {get;}
```

The benefit of using the prompt over the online documentation is that you get to display the value of the properties. This is useful for objects whose property names are not obviously clear at first glance. As you may already know, just like with traditional PowerShell, only a selection of properties is displayed when running a cmdlet like *Get-VM*. This behavior is dictated by special XML files in PowerCLI.

In order to display all the properties of an object, append the " | *select* * " to the cmdlet or variable. Below, you can see the values of the properties shown in the *Get-Member* output. Of course, you will not see the methods as these are actions.

> **Note:** Remember that select as shown above is the shortened version of Select-Object.



This screenshot also depicts the warning messages about object deprecation. You can turn these messages off using:

```
Set-PowerCLIConfiguration -DisplayDeprecationWarnings $false
```

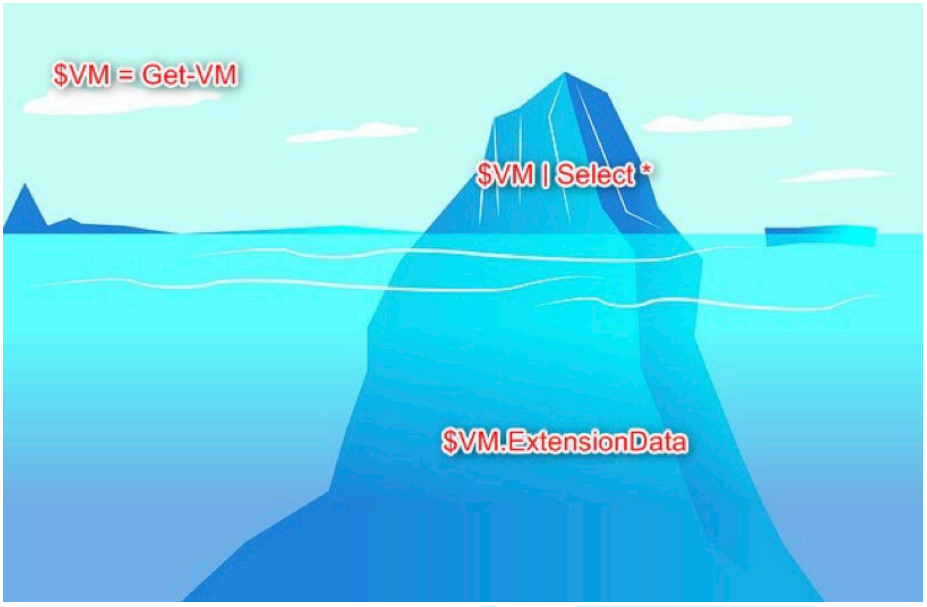
RETRIEVE ADVANCED DATA ON VSPHERE OBJECTS (EXTENSIONDATA)

As demonstrated previously, the output obtained when running a PowerCLI cmdlet such as Get-VM contains a nicely formatted set of properties that most people will be interested in about a VM, and you get more properties when you pipe it in "| *select* *". But that's not all - as you may have realized not everything is available at first glance. Those cmdlets only return a fraction of the data available for a VM.

```
PS C:\> get-vm test-restore | Get-Member

   TypeName: VMware.VimAutomation.ViCore.Impl.V1.VM.UniversalVirtualMachineImpl

Name                        MemberType Definition
----                        ---------- ----------
ConvertToVersion            Method     T VersionedObjectInterop.ConvertToVersion[T](
Equals                      Method     bool Equals(System.Object obj)
GetClient                   Method     VMware.VimAutomation.ViCore.Interop.V1.VIAuto
GetConnectionParameters     Method     VMware.VimAutomation.ViCore.Interop.V1.VM.Rem
GetHashCode                 Method     int GetHashCode()
GetType                     Method     type GetType()
IsConvertableTo             Method     bool VersionedObjectInterop.IsConvertableTo(t
LockUpdates                 Method     void ExtensionData.LockUpdates()
ObtainExportLease           Method     VMware.Vim.ManagedObjectReference ObtainExpor
ToString                    Method     string ToString()
UnlockUpdates               Method     void ExtensionData.UnlockUpdates()
CoresPerSocket              Property   int CoresPerSocket {get;}
CustomFields                Property   System.Collections.Generic.IDictionary[string
DatastoreIdList             Property   string[] DatastoreIdList {get;}
DrsAutomationLevel          Property   System.Nullable[VMware.VimAutomation.ViCore.T
ExtensionData               Property   System.Object ExtensionData {get;}
Folder                      Property   VMware.VimAutomation.ViCore.Types.V1.Inventor
FolderId                    Property   string FolderId {get;}
Guest                       Property   VMware.VimAutomation.ViCore.Types.V1.VM.Guest
GuestId                     Property   string GuestId {get;}
HAIsolationResponse         Property   System.Nullable[VMware.VimAutomation.ViCore.T
HardwareVersion             Property   string HardwareVersion {get;}
HARestartPriority           Property   System.Nullable[VMware.VimAutomation.ViCore.T
Id                          Property   string Id {get;}
MemoryGB                    Property   decimal MemoryGB {get;}
MemoryMB                    Property   decimal MemoryMB {get;}
Name                        Property   string Name {get;}
Notes                       Property   string Notes {get;}
NumCpu                      Property   int NumCpu {get;}
PersistentId                Property   string PersistentId {get;}
PowerState                  Property   VMware.VimAutomation.ViCore.Types.V1.Inventor
ProvisionedSpaceGB          Property   decimal ProvisionedSpaceGB {get;}
ResourcePool                Property   VMware.VimAutomation.ViCore.Types.V1.Inventor
ResourcePoolId              Property   string ResourcePoolId {get;}
Uid                         Property   string Uid {get;}
UsedSpaceGB                 Property   decimal UsedSpaceGB {get;}
VApp                        Property   VMware.VimAutomation.ViCore.Types.V1.Inventor
Version                     Property   VMware.VimAutomation.ViCore.Types.V1.VM.VMVer
VMHost                      Property   VMware.VimAutomation.ViCore.Types.V1.Inventor
VMHostId                    Property   string VMHostId {get;}
VMResourceConfiguration     Property   VMware.VimAutomation.ViCore.Types.V1.VM.VMRes
VMSwapfilePolicy            Property   System.Nullable[VMware.VimAutomation.ViCore.T
```

It’s already a pretty good chunk of properties but there is a wealth of additional information in the *ExtensionData* property which is different from the others (highlighted in yellow). That property offers a convenient way of interacting with the underlying API and returns a lot more methods and properties. In Use Cases we will stick to properties, but we will experiment with more later.

> **Note:** The Get-View cmdlet returns the same results as the ExtensionData property, but we won't get into the details of it as it is a more advanced topic.



To access this “hidden” information, display the content of the *ExtensionData* property on a VM object. The properties that make up the list contain categories in which you will find other sub-categories etc. Using this process, you have access to information such as hardware devices, file types locations, guest related information retrieved by VMware tools, and more! Some might require a little more digging than other to find, but it is easy enough to browse through the properties.

CHAPTER 4: WRITING SCRIPTS

Now that we introduced PowerCLI and started using it, let's take a small break to talk about script quality. When writing scripts, regardless of the language used, it is important to ensure that it is clean, efficient, flexible and as short as possible. Should you fail to do so, you will make it harder for others and your future self to understand your scripts and ensure consistency. I will cover a few topics here that may not apply to everyone but are considered best practices. You will find your own style of scripting and figure out what works best for you as you gain more experience in this area but it's important to consider the following aspects.

SHORTER IS BETTER

One thing that makes a script enjoyable to read is its optimization. Try to avoid repeating the same operation several times. For instance, you can usually make it shorter and more optimized by using a loop that specifies said operation only once. Start by writing the script so it does everything you want it to do, and then go through the code and look for opportunities to optimize it, hence making it cleaner. It might take a bit more time when writing the code but you'll be thankful you did it. Trust me!

NO HARDCODED VALUES

This one has been repeated many times already, but it's very important. **Scripts should always avoid hard-coded values.** It makes them difficult to maintain. It is sure to break any time something is changed in the environment, and no one will remember there is this one script that uses the old value.
Now sometimes you may need to specify values inside a script for whatever reason, and if you have to do it, there are ways the minimize its impact.

- Define all the variables at the top of your script. The rest of the script should **only** use variables to avoid the treasure hunt when something breaks.
- If you write a library of scripts/functions spread across several files, it might be a good idea to consolidate all your statically defined variables into a csv or xml file that is imported at the start of every script. That way when you change a value, you will change it globally.
- Populate variables dynamically as much as possible through the use of vSphere tags, naming conventions, etc. The point is to design your script so it survives changes in the environment.

ERROR HANDLING

This part is maybe the least fun of script writing but it is a necessary evil. Including error-handling in your functions will make them a lot safer to use and easier to debug. This can be achieved in a number of ways but the "Try, Catch, Finally" triad is probably the most convenient one. This method is used in the first of the use cases in this document.

PSSCRIPTANALYZER

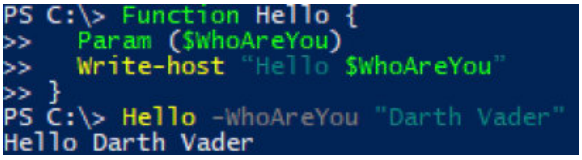
If you really want to make sure your scripts follow Microsoft's guidelines, you can use PSScriptAnalyzer to check the quality of your code. I usually don't go that far but it is there if you need it.

FUNCTIONS

One way of making your script leaner and cleaner is through the use of functions. Similar to a subroutine or a procedure in other programming languages, a function is a bit of code that can take parameters (arguments) and return some results. Writing a function may seem daunting at first, but you'll soon get used to it. Some people think it's a complicated programmer technique when it's actually very easy and satisfying to do.

To illustrate my point, here is a function:

```
Function Hello {
  Param ($WhoAreYou)
  Write-host “Hello $WhoAreYou”
}
```



This is obviously a silly example as it serves no purpose but the reality is not too far from it to get started with functions.

WHY USE FUNCTIONS?

Functions are incredibly useful in a variety of cases:

HELP REDUCE DUPLICATE CODE (REMEMBER, SHORTER IS BETTER)

If the same section of an operation is repeated several times throughout a script, these can be made into a function that is called in one line instead of writing all the operations again. Should a change need to be made to this section, amending the content of the function will update it everywhere in the script, hence reducing the risk of forgetting one of them during a script update.

DECOMPOSING COMPLEX MONOLITHIC SCRIPTS INTO SMALLER CHUNKS

By placing each main activity inside a function that is parameterized and tested, the execution script becomes an orchestration script that calls the functions one after the other, making it shorter and easier to debug.

SAVING TIME

PowerShell and PowerCLI offer a wide selection of cmdlets but some operations might not be covered by them or you might want to have the possibility to make something quicker and easier to do. For instance, you could have a function that decommissions a datastore properly by un-mounting it and detaching the LUN on all the hosts. By putting them into your own PowerShell module, you can have your own functions available in the prompt! The possibilities are endless!

FUNCTIONS BEST PRACTICES

There is a lot to say about functions, in fact they could be the topic of a book all by themselves! For that reason, we will only skim the surface of what you can and should do with them. For more information, there is a plethora of great content online that talks about them in greater detail. Below are a few guidelines to write clean functions and help others, as well as your future-self, understand your work.

USE PARAMETERS, AND USE THEM WELL

For cleaner functions, always use the *param()* block and put the parameter segment below the title, not on the same line. Doing so will allow you to add attributes to your parameters. Attributes are a great way to lock down your function to ensure it is used properly. In this manner, functions can even do some error handling for you.

Example of how the param() block should be used:

```
Function Convert-DSToCanonical {
param(
    [Parameter(Mandatory = $True,ValueFromPipeline=$True)]
    [VMware.VimAutomation.ViCore.Impl.V1.DatastoreManagement.DatastoreImpl[]]
    $Datastore
)
```

USE PARAMETER ATTRIBUTES

Attributes apply conditions to parameters to ensure they is used the way they're intended.

Lots of attributes are available that deal with a variety of needs. Among them, you can lock a parameter to a specific type, make it mandatory or not, allow only a certain set of values, allow one or multiple values, set a default value, validate a parameter with a bit of code, give it parameter aliases, prevent sets of parameters to be used together, and much more! Find an extensive list on Microsoft Docs.

Example of a few parameter attributes:

```
[CmdletBinding(DefaultParameterSetName=1)]
param(
    [Parameter(Mandatory=$true,ValueFromPipeline =
$True,position=0)]
    [string]
    $Destination,

    [parameter(position=1)]
    [ValidateRange(1,65535)]
    [int]
    $Port,

    [parameter(parametersetname=2)]
    [Alias('t')]
    [switch]
    $Continuous,

    [parameter(parametersetname=1)]
    [Alias('n')]
    [int]
    $Count = 4
```

USE THE PROPER NOMENCLATURE

It is always best to try and stick to Microsoft's guidelines to make your functions as intuitive as possible.

Follow the verb-noun nomenclature and use approved verbs *(Get-Verb)*. That way your scripts will have a more “PowerShell feel” to them. For instance, use *Get-VMHost* instead of *Retrieve-VMHost*.

Try to use common variable names like *$VMHost* instead of *$ESX*.

CODE INDENTATION

Using indentation greatly helps in understanding the structure of the code and increases clarity. Any script that is not correctly indented will require extra effort to make sense of what it does and makes troubleshooting more difficult. And please, avoid mile-long one-liners. Don't be that guy.

COMMENT YOUR CODE

Here is another one that has been repeated since well before PowerShell existed. Include as many comments as possible to help others, or your future self, understand your code. Note that you can even create a help page for your functions.

Example of a help block for a custom made function.

```
PS C:\> help Invoke-Watch -full

NAME
    Invoke-Watch

SYNOPSIS


SYNTAX
    Invoke-Watch [-ScriptBlock] <ScriptBlock> [-Interval <Int32>] [<CommonParameters>]


DESCRIPTION
    Simulates Linux's watch command.
    invoke-watch runs command repeatedly, displaying its output (the first screenfull).
    This allows you to watch the program output change over time. By default, the
    program is run every 2 seconds; use -n or -interval to specify a different interval.


PARAMETERS
    -ScriptBlock <ScriptBlock>

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    -Interval <Int32>

        Required?                    false
        Position?                    named
        Default value                2
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <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).

INPUTS

OUTPUTS

    -------------------------- EXAMPLE 1 --------------------------

    PS C:\>invoke-watch {get-process | sort CPU -Descending | select -first 5 | ft}

    Shows the top 5 processes on CPU usage



    -------------------------- EXAMPLE 2 --------------------------

    PS C:\>{Get-Volume | ft -au} | invoke-watch -n 5

    Shows volumes with a refresh every 5 seconds
```

RETURN OBJECTS

If your function is going to return something after execution, make it PowerShell object and not strings. Returning object will allow you to use them for other purposes. And in the same spirit as the nomenclature, use common names for the property names of the objects returned (like *VMHost*). That way, you can make use of the *PipelineByPropertyName* capability of PowerShell.

CHAPTER 5: HOW TO USE SCHEDULED TASKS AND HTML REPORTS

I grouped these two sections together because they often work hand in hand. I usually write a script that produces a report in HTML format and then store it on a web server or send it via email.

That script is run by a scheduled task so I have an updated version available.

TASK SCHEDULING

We have all been creating scheduled tasks for a long time now and everyone should be familiar with it. Here we are covering a nifty convenient way to execute PowerShell scripts in scheduled tasks. Those scheduled tasks can be used for many things like a simple snapshots listing, an advanced VM inventory or a more complicated SRM cleaning script, for instance.

PREPARATION

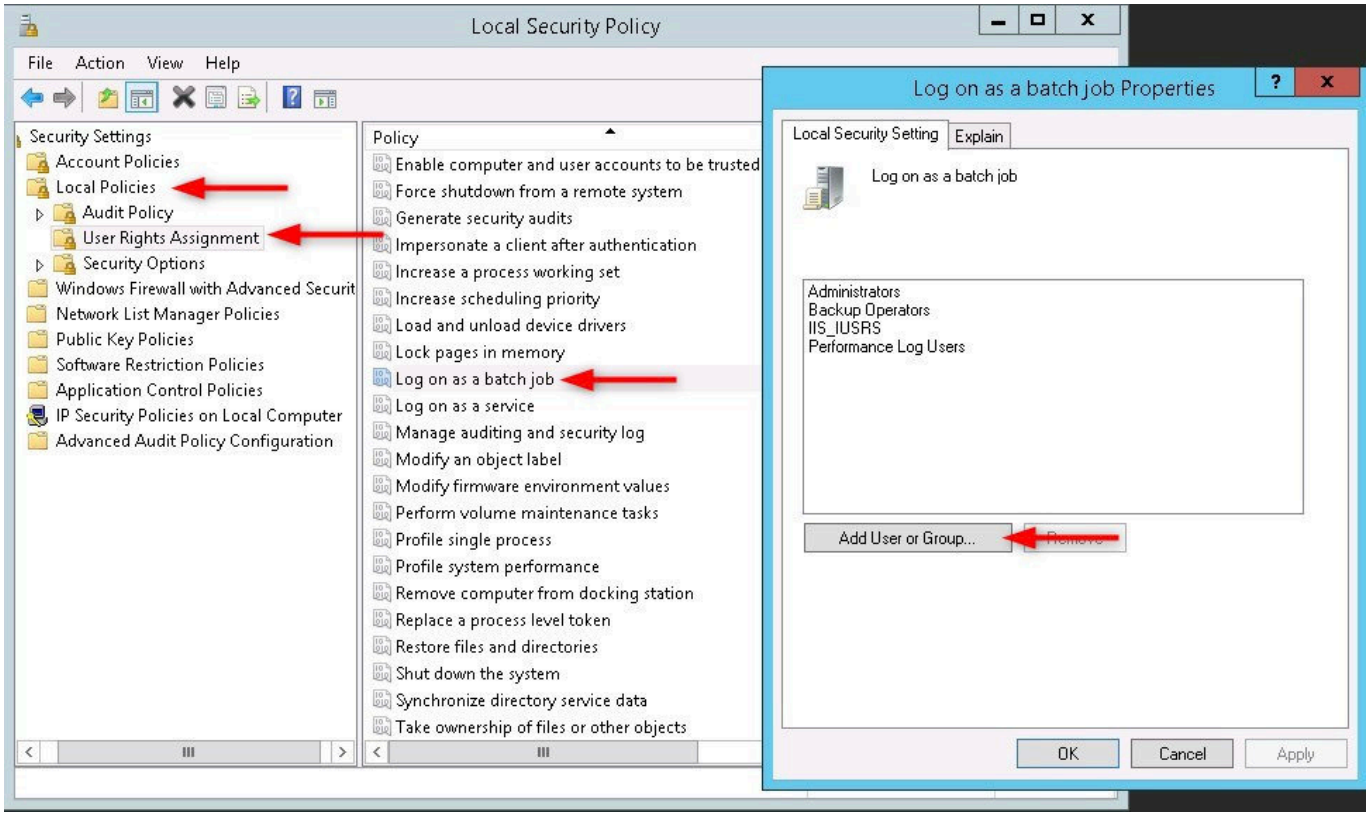
DEFINE WHERE THE SCRIPTS WILL BE RUNNING

- It needs to be a Windows machine that has at least access to vCenter Server on port 443
- Install the PowerCLI modules at the local machine level or user profile level of the service account used to run the scripts

CREATE A SERVICE ACCOUNT FOR THE SCHEDULED TASK

- A domain user on the AD is better but local users also work. We'll name it **svc-task-ps** in this example
- Open the Local Security Policy console by typing secpol in windows search
- Expand Local Policies > User Rights Assignment

- Locate "*Log on as a batch job*" and open its properties
- Add the user or group you created previously



- Launch a PowerShell session as this user and set the execution policy if necessary

  (c.f. Installing PowerCLI)

CREATE A FOLDER TO STORE THE SCRIPTS AND APPLY PERMISSIONS

- Create the folder anywhere on the server that makes sense to you (C:\Scripts for instance)
- Apply write permission on this folder to **svc-task-ps**

APPLY VSPHERE PERMISSIONS

- Log in the vSphere web client and select the vCenter object
- Apply permissions to **svc-task-ps**. Try and restrict them to the bare minimum. For instance, if the scripts are only doing reports, you can give the user "Read-Only" access permission

TEST USER

- Open a PowerShell session as svc-task-ps on the machine that will run the scheduled tasks
- Ensure you can connect to vCenter:

```
Connect-VIServer vcsa.test.local
```

- Make sure you can perform some of the command you will need in your scripts

INSTALLING THE SCRIPTS

The idea here is to make scheduling scripts as flexible and easy as possible. To achieve this, we have a batch file that runs a PowerShell script of the same name in the same folder and prints its execution in a log file. It is convenient because whenever you need to schedule a new script, just copy this batch file in the new folder, rename it like the ps1 file and you're done!

STORE THE POWERSHELL SCRIPT IN THE DEFINED LOCATION: C:\Scripts\ListVMs.ps1

This test script stores a list of the VMs in a csv file:

C:\Scripts\ListVMs.ps1

```
$VCENTER = "vcsa.test.local"
$CSVFile = C:\Scripts\VMList.csv
Connect-VIServer $VCENTER
Get-VM | Export-Csv $CSVFile
```

STORE THE BATCH FILE IN THE SAME LOCATION: C:\Scripts\ListVMs.bat

This script does the following:

1. Changes directory to the location of the script (*%~dp0*)

2. Creates a variable with the date of today as yyyymmdd

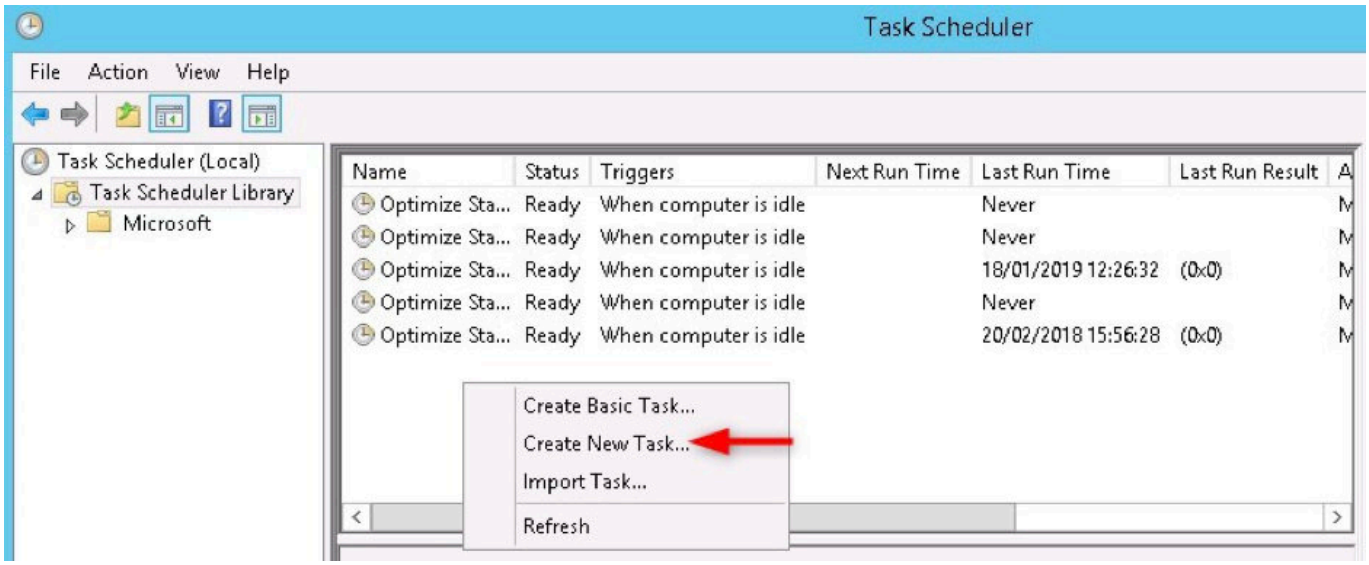
3. Executes a ps1 file with the same name as this bat file (*%~n0*) in PowerShell and sends the output to a log dated file (>> *%~dp0\%~n0-%runDate%.log*)

C:\Scripts\ListVMs.bat

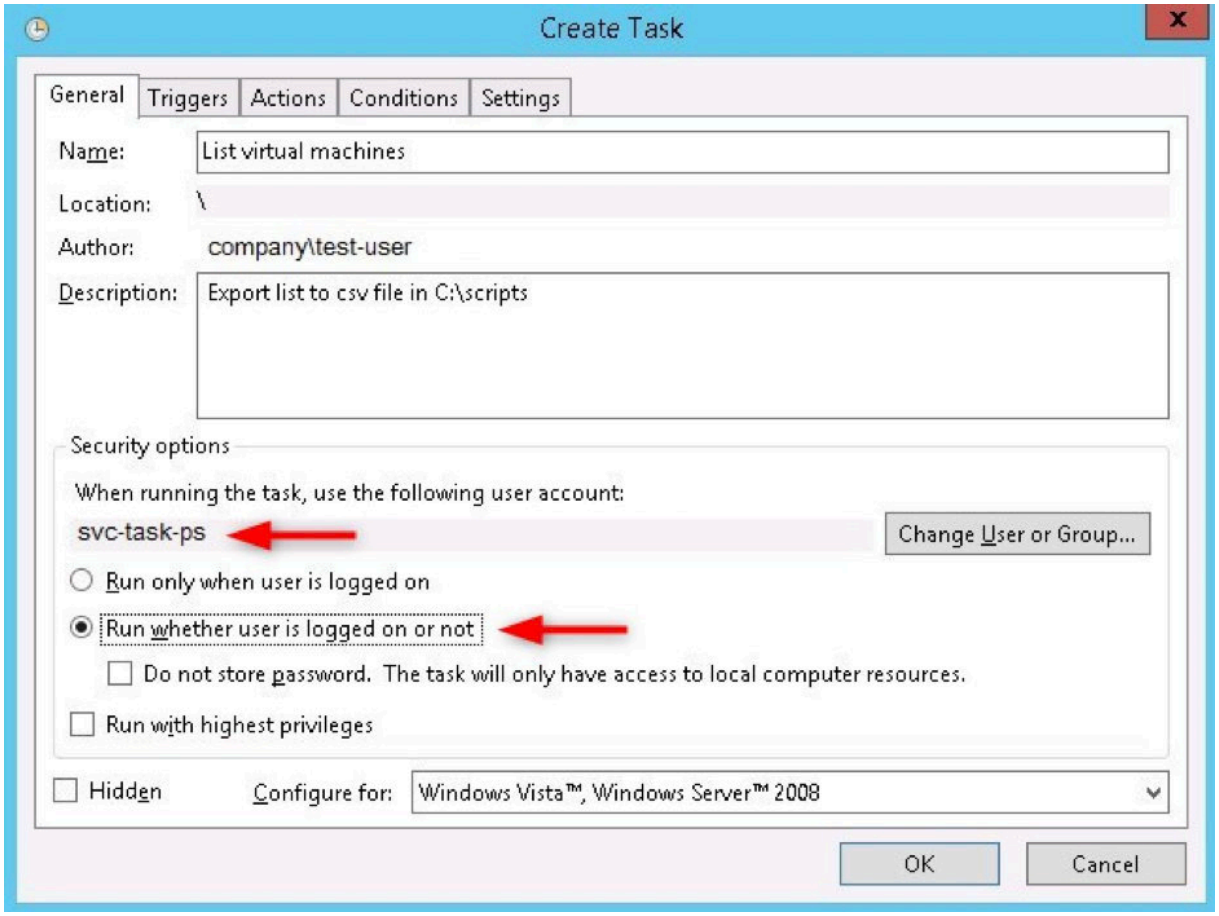
```
CD %~dp0

SET runDate=%DATE:~6,4%_%DATE:~3,2%_%DATE:~0,2%

C:\Windows\System32\cmd.exe /c powershell.exe -noninteractive -
noprofile -file %~dp0\%~n0.ps1 >> %~dp0\%~n0-%runDate%.log
```

CREATE THE SCHEDULED TASK

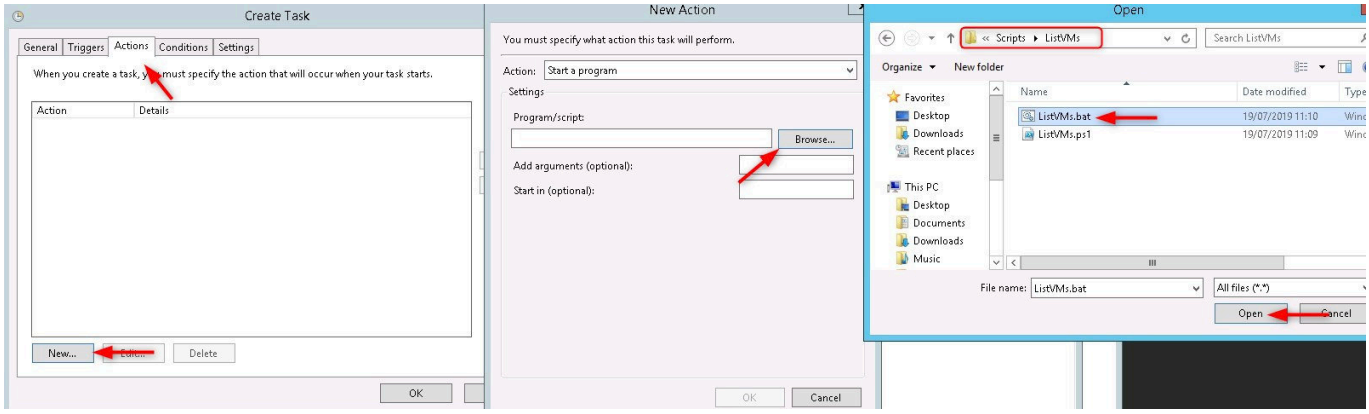
- Open the task scheduler and create a new task

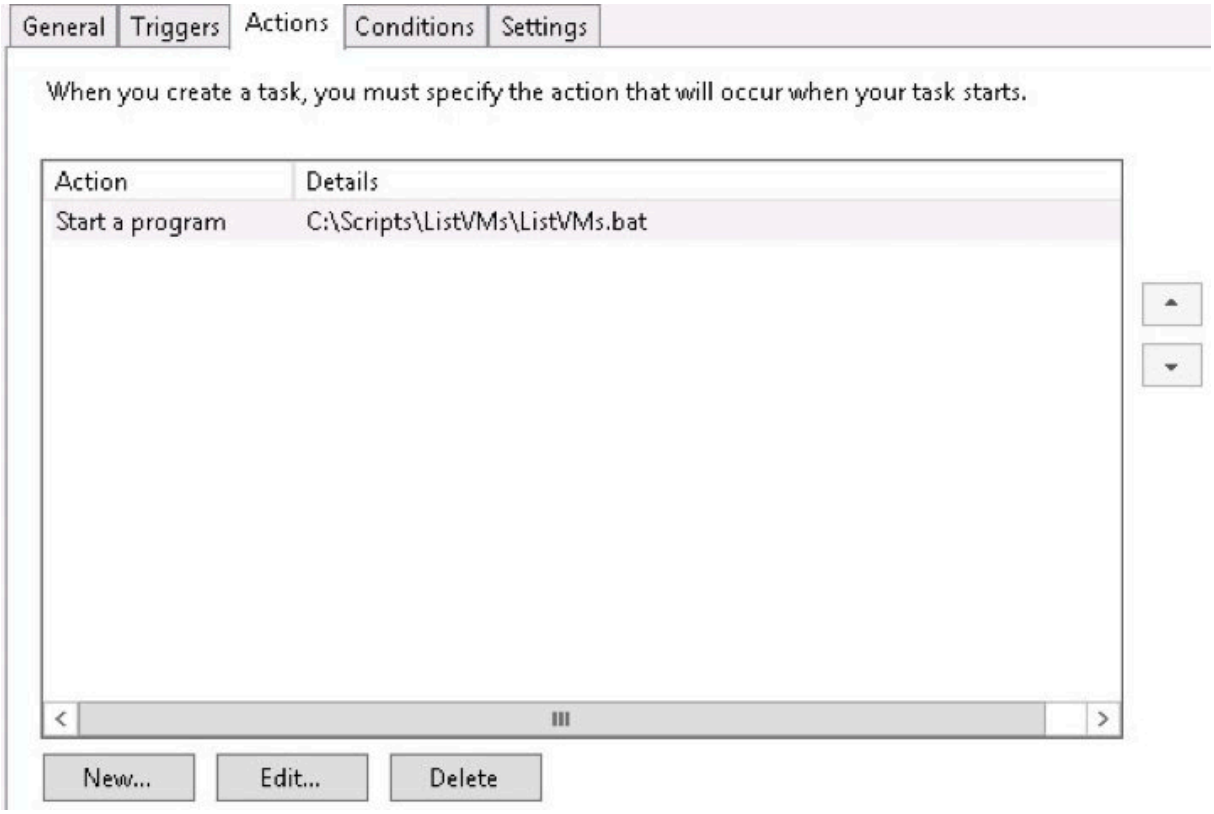


- Select the task user we created earlier and check "**Run whether user is logged on or not**"



- Configure the **Trigger** tab as you want. You can leave it empty for this test as we can run the task manually
- In the **Action** tab leave the default settings and select the batch file





- Leave the rest of the tabs to their default settings unless specifically needed. Click *OK* in the *General* tab to confirm and run the task to make sure that it works

HTML REPORTING

Reporting is an important part of managing and automating an environment. Not only does it give you a wider view, it also takes jobs off your hands by scheduling it. You won't forget small tasks and you can use your time on more interesting or important concerns. While most of the vSphere monitoring can be done through vROps (vRealize Operations Manager), many companies don't have the budget for it or simply want quick and dirty email reports.

PowerShell offers a convenient way to convert objects into HTML code with a single cmdlet "*ConvertTo-HTML*". Once again, we will take the example of a VM inventory that we store as an HTML file on a web server or send via email.

PRODUCING THE HTML REPORT

Let's start from the beginning, first we need to retrieve our VM inventory, then convert it to HTML, then process it for email, web server or both.

- Retrieve the list that should make up the report

```
Connect-VIServer vcsa.test.local

$VMList = Get-VM | where powerstate -eq poweredon | select
name,vmhost,PowerState,memoryGB,numcpu,guest,version
```

- Convert the list to HTML

> **TIP:** Add style to your HTML to create borders in your reports. Write your CSS in a $Style variable that you pass in the *-Head* parameter of the *ConvertTo-Html* cmdlet. For instance, you could use this to add a background color to the front row according to the status of an object.

```
$Style = "<style>table, th, td {border: 1px solid;}</style>"

$Html = $VMList | ConvertTo-HTML -Head $Style
```

There are many other interesting parameters in *ConvertTo-Html*. Have a look at its documentation if you want to explore this further.

PROCESSING THE REPORT

EMAIL REPORT

- I suggest you populate a set of variables with the information relative to your email environment. That way you can re-use it for your next script and only change some values.

```
$Email = @{
    Subject = "List VMs"
    From = "$Env:Computername@test.local"
    To = test-user@test.local
    SmtpServer = srv-smtp.test.local
    Body = $Html
    BodyAsHtml = $True
}
```

Note 1: Here we create an object containing the parameters of our Send-MailMessage cmdlet. This is called **Splatting** and gives flexibility when adding/removing parameters in a clean fashion.

Note 2: The From field is dynamically populated with the name of the server as address prefix.

- All that's left to do now is to send the email

```
Send-MailMessage @Email
```

WEB SERVER

If you want to have the report available at any time in your web browser, all you need to do is to export it to the web server's www folder with the .html extension. Note that in order to achieve this, the user running the task must have write permissions to the www folder. Also, I suggest you add the date and time of the execution in the web page to know at which point in time this information was retrieved.

- Add the date at the top of the report

```
$Style = "<style>table, th, td {border: 1px solid;}</style>"

$Html = $VMList | ConvertTo-HTML -Head $Style -PreContent
"<h1>$(get-date)</h1>"
```

- Export the Html file

```
$File = "\\srv-web\www\ListVMs.html"

$Html | Out-File $File -Force
```

Not the most beautiful report as you can see but it gets the job done. Once you have a framework down for nicer reports, you can start experiencing with more advanced scripts and maybe build an index page with links to the reports, a great exercise.

07/19/2019 14:12:46

Name	VMHost	PowerState	MemoryGB	NumCpu	Guest	Version
srv-vm-01	ESX-Host-1	PoweredOn	10	2	srv-vm-01:Microsoft Windows Server 2016 (64-bit)	v13
srv-vm-02	ESX-Host-1	PoweredOn	12	4	srv-vm-02:Microsoft Windows Server 2008 R2 (64-bit)	v13
srv-vm-03	ESX-Host-3	PoweredOn	4	2	srv-vm-03:Microsoft Windows Server 2016 (64-bit)	v13
srv-vm-05	ESX-Host-2	PoweredOn	3	2	srv-vm-05:Microsoft Windows Server 2008 R2 (64-bit)	v13

CHAPTER 6: USE CASES

While a large number of administrative tasks can be achieved using the built-in vCenter alerts and the web client, others can only be done through the use of the API (which PowerCLI leverages). On top of that, some actions may take a painfully long time when performed manually in the vSphere client, when they could be completed in a matter of seconds with PowerCLI.

In this section, we will look at how you can improve your daily admin operations and make the RUN process a little more streamlined.

DEPLOYING VIRTUAL MACHINES FROM A CSV FILE

This use case is probably one of the most frequently requested by companies when it comes to automation. Deploying virtual machines is a day-to-day task, but it can take a lot of time and IT departments are understandably eager to automate this process. Here we are going to show you how to deploy several virtual machines based on the details stored in a CSV file. Keep in mind that this is merely an example - you can go much, much further with automation in PowerCLI.

REQUIREMENTS

These requirements are only true for the purpose of this script because I wanted to keep it as simple as possible. You can make them irrelevant with a few changes to the code.

- At least one Windows template
- At least one OS customization specifications file

- A DRS enabled cluster not set to manual
- A CSV file including the details of the VMs to deploy. The file should include the following columns:
    - VMName
    - Template
    - IP
    - Mask
    - Gateway
    - Dns
    - Cluster
    - Datastore
    - Portgroup

Here is an example of the content of a valid CSV file:

```
VMName,template,IP,mask,gateway,dns,cluster,datastore,portgroup
test-vm-1,W2016Template,10.1.1.5,255.255.255.0,10.1.1.254,10.1.1.1,
Cluster_test,vsandatastore,Test-PG
test-vm-2,W2016Template,10.1.1.6,255.255.255.0,10.1.1.254,10.1.1.1,
Cluster_test,vsandatastore,Test-PG
```

THE SCRIPT EXPLAINED

The script below does exactly what we just mentioned. I used comments (in green) to define sections ranging from A to H that will be explained independently so you can refer to the script.

Note that this script is sequential, meaning it does things one by one. You can make this kind of scripts run faster by using jobs running concurrently but this is out of the scope here.

```
######################### SECTION A
# Make errors to be terminating. Otherwise some won't be caught.
$ErrorActionPreference = "Stop"
Try {
# Specify the location of the CSV file.
$CSVFile = "C:\Users\test-user\Desktop\DeployVM.csv"
$CSVVm = Import-Csv $CSVFile
foreach ($CSVRow in $CSVVm) {

    ######################### SECTION B
    # Verifying that the input is correct and that the VM name is
not already in use.
    $VmTemplate  = Get-Template $CSVRow.Template
    $VmPortgroup = Get-VirtualPortGroup -Name $CSVRow.Portgroup
    $VmDatastore = Get-Datastore $CSVRow.datastore
    $VmCluster   = Get-Cluster $CSVRow.Cluster | where
DrsAutomationLevel -ne "manual"
    if (Get-VM $CSVRow.VMName -ErrorAction SilentlyContinue) {Throw
"A virtual machine with name $($CSVRow.VMName) already exists"}

    ######################### SECTION C
    # Cloning existing custom specs to a temporary one
    $CustomSpec = New-OSCustomizationSpec -OSCustomizationSpec Test-
win -Name (New-Guid | select -ExpandProperty guid) -Type
NonPersistent

    ######################### SECTION D
    # Configuring the network settings in the custom specs
    $CustomSpec | Get-OSCustomizationNicMapping | Set-
OSCustomizationNicMapping -IpMode UseStaticIP -IpAddress $CSVRow.IP
-SubnetMask $CSVRow.mask -DefaultGateway $CSVRow.gateway -Dns
$CSVRow.dns

    ######################### SECTION E
    # Deploying the VM
    $NewVM = New-VM -Template $VmTemplate -OSCustomizationSpec
$CustomSpec -ResourcePool $VmCluster -Datastore $VmDatastore -Name
$CSVRow.VMName

    ######################### SECTION F
    # Connecting the correct portgroup
    $NewVM | Get-NetworkAdapter | Set-NetworkAdapter -StartConnected
$true -Confirm:$false -NetworkName $CSVRow.portgroup

    ######################### SECTION G
    # Starting the VM
    $NewVM | Start-VM
}
```

```
} CATCH {

    ######################### SECTION H
    Write-Error $_.Exception -ErrorAction stop
}
```

SECTION A

As part of the error handling to implement, we want the script to stop should any error occur. In order to achieve this ,we use the *TRY, CATCH* blocks that offer a convenient and clean solution. However, non-terminating errors are not caught by the CATCH block and the cmdlets we use here generate some for input check purposes. The *$ErrorActionPreference* variable is a preference variable that determines how PowerShell responds to a non-terminating error. In this section we set the variable to "Stop" to ensure that these errors are caught.

Then we use the *Import-CSV* cmdlet to import the content of the CSV file as a collection of PowerShell objects that we store in the *$CSVVm* variable.

```
PS C:\Users\xavier> Import-Csv .\DeployVM.csv | Format-Table

VMName    template      IP       mask          gateway     dns      cluster      datastore     portgroup
------    --------      --       ----          -------     ---      -------      ---------     ---------
test-vm-1 W2016Template 10.1.1.5 255.255.255.0 10.1.1.254 10.1.1.1 Cluster_test vsandatastore Test-PG
test-vm-2 W2016Template 10.1.1.6 255.255.255.0 10.1.1.254 10.1.1.1 Cluster_test vsandatastore Test-PG
```

Finally, we start a *foreach{}* block that loops through every record of the CSV file (shown above).

SECTION B

This section serves two purposes. It retrieves vSphere inventory objects for later use and breaks the execution if the object cannot be found. That way we can ensure that the user didn't make a typo in an object name or chose a VM name already in use. Because it is located inside the *TRY{}* block, any error that occurs will be caught by the *CATCH{}* block (more on that in section H).

SECTION C

Here we are essentially cloning an existing customization specifications file into a temporary one. The type *NonPersistent* means that the cloned custom specs are only available in the current PowerCLI session; it doesn't appear in the UI and is destroyed when the session is closed. Because it is temporary, we give it a random name using the *New-Guid* cmdlet.

SECTION D

We now need to configure the cloned custom specs using the input from the CSV file. In order to do so, we have to use yet another type of object with the *Get/Set-OSCustomizationNicMapping* cmdlet. We then use the input of the CSV file to set the properties.

SECTION E

The ground work is done so we can now deploy the VM with the cloned custom specs, template and resources configured in the CSV (datastore and cluster). The prompt will stay on hold for the duration of the deployment with a progress bar before moving on to the next step. Once the VM is created the inventory object of the new VM is stored in the *$NewVM* variable.

SECTION F

By default, the VM created sits on the same network portgroup as the template it originates from. Any hardware changes (CPU, Memory, disk, etc.) should be done here, including the portgroup that was part of the CSV input.

SECTION G

The VM is now here and configured and it is time to power it on. Once the VM boots up, it will start the guest customization following the cloned custom specs we created earlier. So be patient as it usually takes several minutes to complete.

After vCenter initiated the VM power on, the script moves on to processing the next line in the CSV file and goes back to Section B.

SECTION H

If any error occurs within the *TRY{}* block, it is intercepted and processed in the *Catch{}* block. Here the $_ variable contains the error encountered which *Exception* property contains the description of the problem. Ending up here means the script will display the error and stop its execution.

There are different schools of thought as to how error handling should be performed. Some prefer to break the current loop and move on to the next record to process, others like me prefer to stop everything so I can do my debugging and check my inputs. No right or wrong answer here, a good old "It depends" as usual.

> **Note:** If you'd like to see another PowerCLI Deployment Sample, we have a great article with a free script on the Altaro DOJO.

MANAGE VSPHERE TAGS

It wasn't easy selecting which use case to cover and which to leave aside as so much can be done in PowerCLI. However, I couldn't leave out vSphere tags because they offer so much versatility and can greatly simplify automation. They are similar to attributes, like metadata you apply to vSphere objects in the inventory, allowing you to create organizational groups that would otherwise not exist (Company department, site locality for metroclusters, role, etc.).

Tags that are related to each other are assigned to a category that can only contain one type of object tags (virtual machine, host, datastore, etc.). Categories have a cardinality property that defines whether more than one tag in the category can be applied to an object.

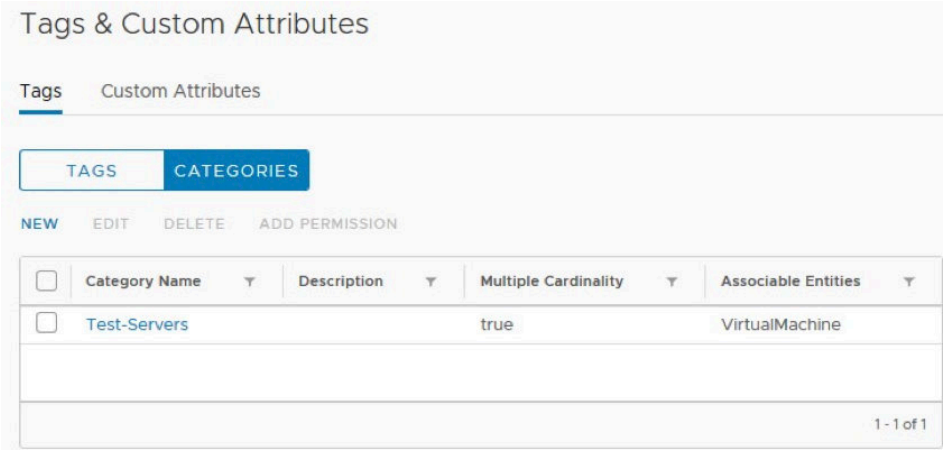
CREATE A TAG CATEGORY

The Entity types you can use for a tag category are the following: *All (default), Cluster, Datacenter, Datastore, DatastoreCluster, DistributedPortGroup, DistributedSwitch, Folder, ResourcePool, VApp, VirtualPortGroup, VirtualMachine, VM, VMHost.*

When setting the Cardinality, "*Single*" means that only a single tag from this category can be assigned to a specific object at a time as opposed to "*Multiple*".

You can also use the *Get-TagCategory* cmdlet to list them:

```
PS C:\> New-TagCategory -Name "Test-Servers" -Cardinality Multiple -
EntityType VirtualMachine

Name                                           Cardinality Description
----                                           ----------- -----------
Test-Servers                                   Multiple
```



CREATE TAGS

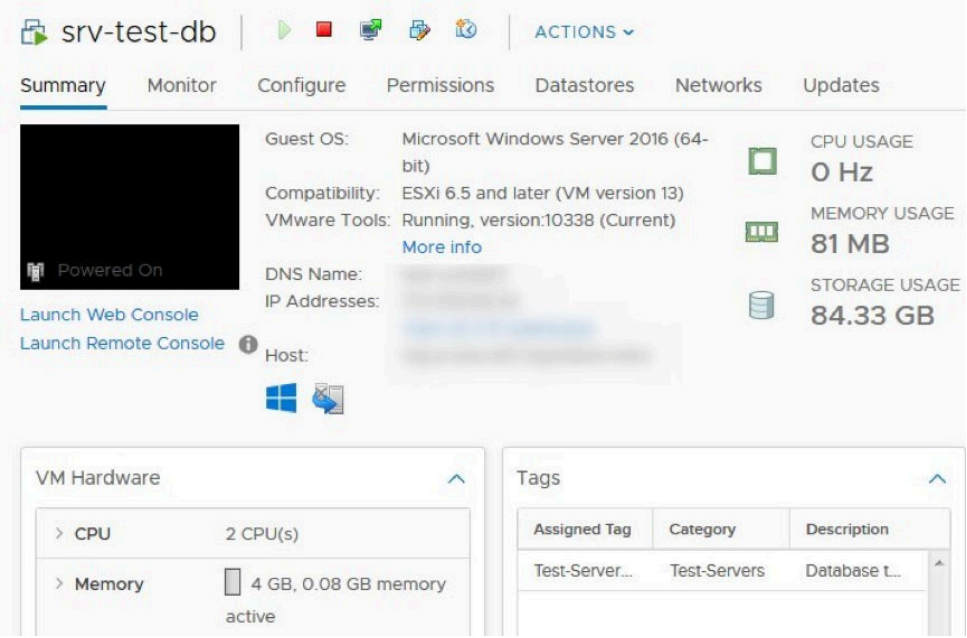
Create as many tags as required and assign them to the category. You can also use the Get-Tag cmdlet to list them.

```
PS C:\> New-Tag -Name Test-Server-DB -Category Test-Servers -Description "Database
test servers"
Name                           Category                       Description
----                           --------                       -----------
Test-Server-DB                 Test-Servers                   Database test servers

PS C:\> New-Tag -Name Test-Server-Web -Category Test-Servers -Description "Web test
servers"

Name                           Category                       Description
----                           --------                       -----------
Test-Server-Web                Test-Servers                   Web test servers
```

ASSIGN THE TAGS TO VMS

Now tag the VMs. As you can see below the VM object can be fed from the pipe to the cmdlet.

```
PS C:\> New-TagAssignment -Tag Test-Server-DB -Entity (get-vm srv-test-db)

Tag                                    Entity
---                                    ------
Test-Servers/Test-Server-DB            srv-test-db

PS C:\> Get-VM srv-test-web | New-TagAssignment -Tag Test-Server-Web

Tag                                    Entity
---                                    ------
Test-Servers/Test-Server-Web           srv-test-web
```

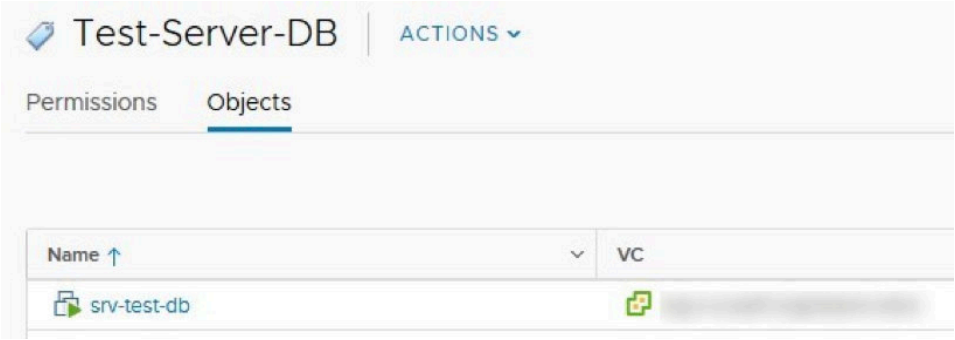


RETRIEVE VSPHERE OBJECTS BASED ON THEIR TAG.

It is now possible to retrieve vSphere inventory based on the tags assigned to them.

The *-Tag* parameter is available on all vSphere object types that can be tagged:

```
PS C:\> Get-VM -Tag Test-Server-DB

Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
srv-test-db          PoweredOn  2        4.000
```



ADD A PORTGROUP ON A STANDARD VSWITCH ON ALL HOSTS IN A CLUSTER

Virtual portgroups in vSphere can be implemented on distributed vSwitches or standard vSwitches. The former lets you create a portgroup quickly on all hosts that are part of the vSwitch. However, standard switches are often used due to license restriction or simply personal preferences. With that in mind, adding portgroups to standard vSwitches on a number of hosts can get slightly frustrating and time consuming.

This can be done a lot faster using PowerCLI instead of the web client.

- Retrieve the vSwitches on which you want to create the portgroup of all host in the cluster:

```
PS C:\> Get-Cluster Cluster_Test | Get-VMHost | Get-VirtualSwitch -Name vSwitch2

Name                           NumPorts   Mtu    Notes
----                           --------   ---    -----
vSwitch2                       9216       1500
vSwitch2                       9216       1500
vSwitch2                       9216       1500
```

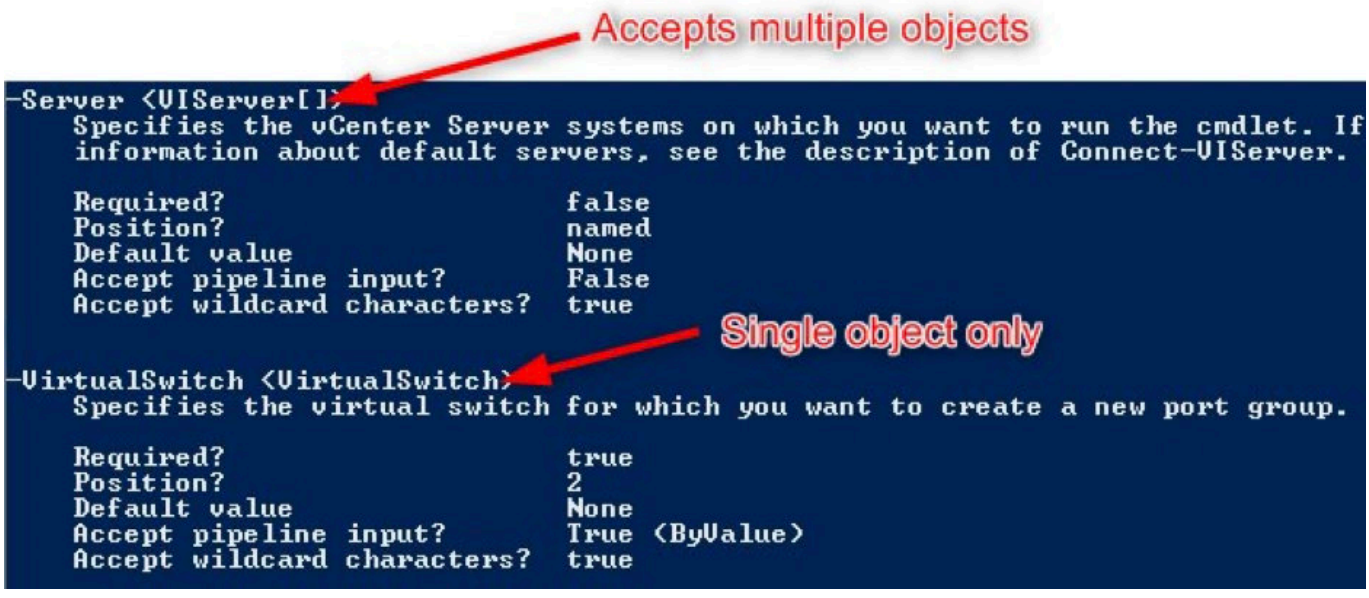
- Create the portgroup with the right vLAN ID:

I stored the vSwitches in a variable to make the command shorter and easier to read.

```
PS C:\> $vSwitches | New-VirtualPortGroup -Name "Test-PG" -VLanId 900
Name                       Key                                VLanId PortBinding NumPorts
----                       ---                                ------ ----------- --------
Test-PG                    key-vim.host.PortGroup-Test-PG 900
Test-PG                    key-vim.host.PortGroup-Test-PG 900
Test-PG                    key-vim.host.PortGroup-Test-PG 900
```

Note: That the -VirtualSwitch parameter does not accept multiple objects. So, you cannot specify an array of values using the named parameter. However, you can use it in the pipeline as it means each record is processed one after the other. You can find in the help page of the cmdlet which object accepts multiple values if there is "[ ]" next to its type.

```
help New-VirtualPortGroup -Parameter *
```



MONITOR VM CPU USAGE IN NEAR REAL TIME

In this use case, we are going to play a little bit with the progress bar capabilities of PowerShell and discover the PowerCLI cmdlet to get performance data. We want to monitor the CPU usage of a VM in near-real time, which is every 20 seconds in vCenter. First, we will learn how to retrieve the correct stats about a VM, and then we will see how to present it graphically.

Some resource usage values like CPU can be retrieved quicker in the *ExtensionData.Summary.QuickStats* property. However, here I will demonstrate the use of the *Get-Stat* cmdlet that includes a lot more information.

RETRIEVE STATS

PowerCLI includes the Get-Stat cmdlet that returns statistical information available on a vCenter server. The *Get-StatType* cmdlet allows you to list all the available counters on an entity.

In our example we want the CPU usage counter.

- Store a VM object in a variable for our tests:

```
$VM = Get-VM test-vm
```

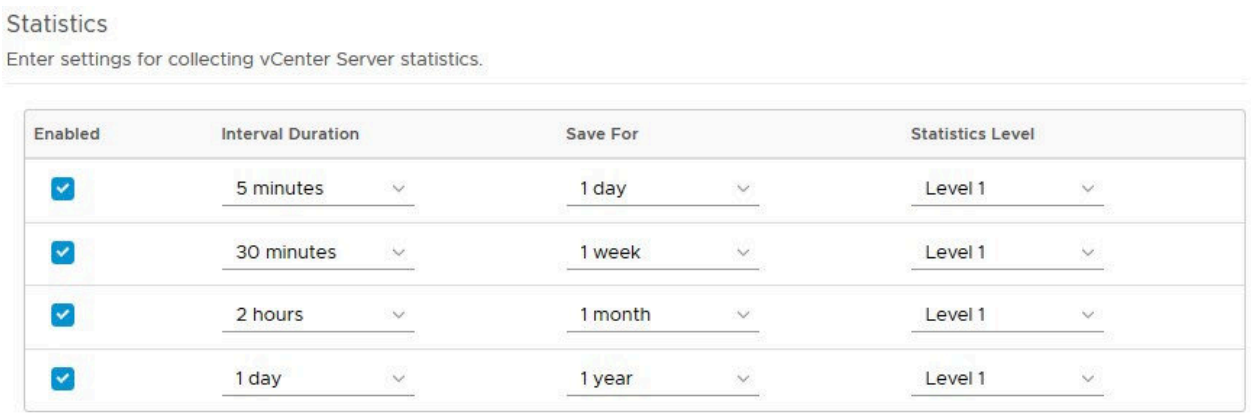
- List the counters that can be retrieved and find the one you need (in this case it is the first one):

```
Get-StatType -Entity $VM

cpu.usage.average
cpu.usagemhz.average
cpu.ready.summation
mem.usage.average
mem.swapinRate.average
mem.swapoutRate.average
mem.vmmemctl.average
mem.consumed.average
mem.overhead.average
disk.usage.average
disk.maxTotalLatency.latest
net.usage.average
```

```
sys.uptime.latest
disk.used.latest
disk.provisioned.latest
disk.unshared.latest
```

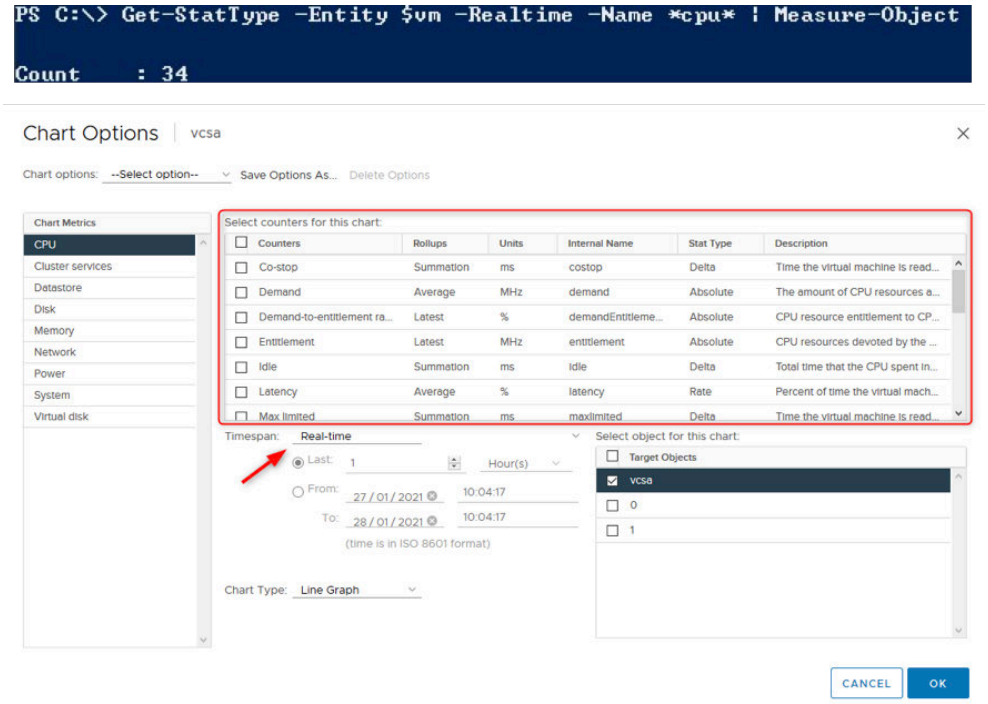
By default, vCenter stores a limited number of counters as the interval increases to save space in the database.

The latest 5 minutes period contains the real-time counters with an interval of 20 seconds.



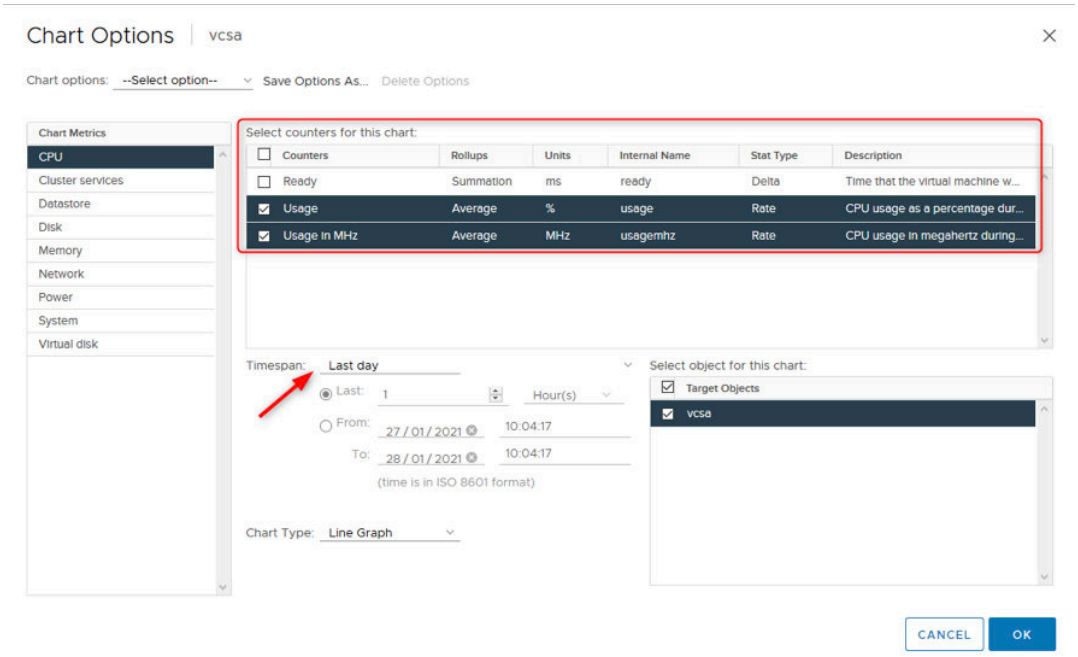
If you add the *-RealTime* switch to the command below you will get a lot more available counters, just like you would in the performance tab of the vSphere Web Client.

REAL-TIME



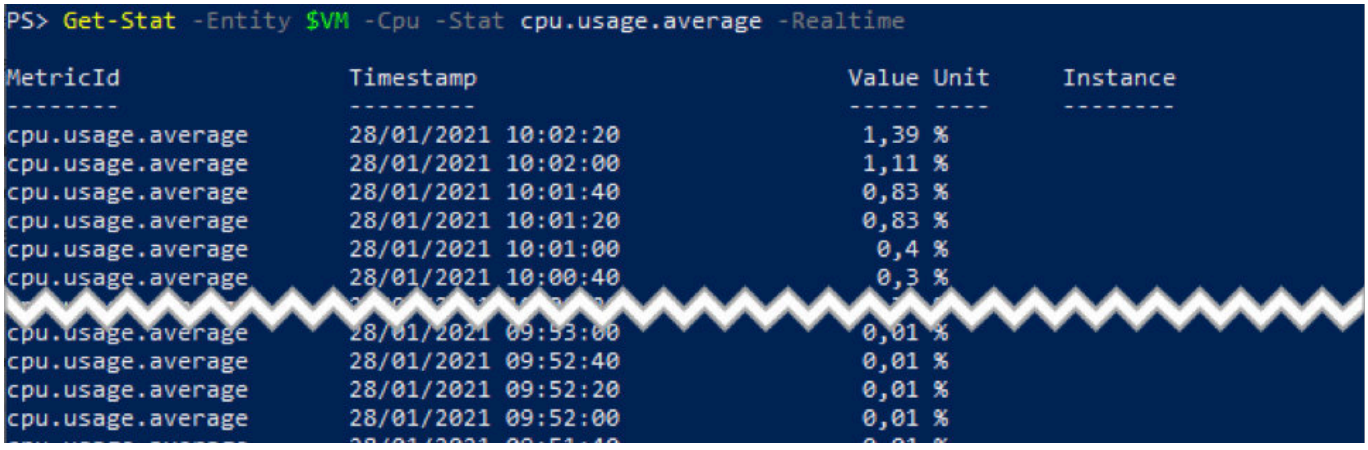
LAST DAY

```
PS C:\> Get-StatType -Entity $vm -Name *cpu* | Measure-Object

Count    : 3
```



- Retrieve the real-time performance data of the VM with this counter:

```
Get-Stat -Entity $VM -Cpu -Stat cpu.usage.average -Realtime
```



A large number of records is returned with an occurrence every 20 seconds. We notice that there are MHz per core (instance number), MHz global (no instance number) and percent global.

- Filter only the last record in percent:

```
get-stat -Entity $VM -Cpu -Stat cpu.usage.average -Realtime
-MaxSamples 1 | where unit -eq "%"
```

```
PS> get-stat -Entity $VM -Cpu -Stat cpu.usage.average -Realtime -MaxSamples 1 | where unit -eq "%"

MetricId             Timestamp                Value Unit     Instance
--------             ---------                ----- ----     --------
cpu.usage.average    28/01/2021 10:03:20       0,92 %
```

PROGRESS BAR

It may look like a gadget to some, but the *Write-Progress* feature of PowerShell is pretty neat to build visual depiction of information inside the prompt. The progress bar is usually used to track installation or script steps, but you can trick it into displaying any kind of values you want formatted as a bar, provided you can make it a percentage. For our particular case, in order to make it act like a real-time monitoring, the command will have to run inside a loop.

- Make a loop that runs every 20 seconds (interval for real-time counters)

```
while ($true) {
 sleep 20
}
```

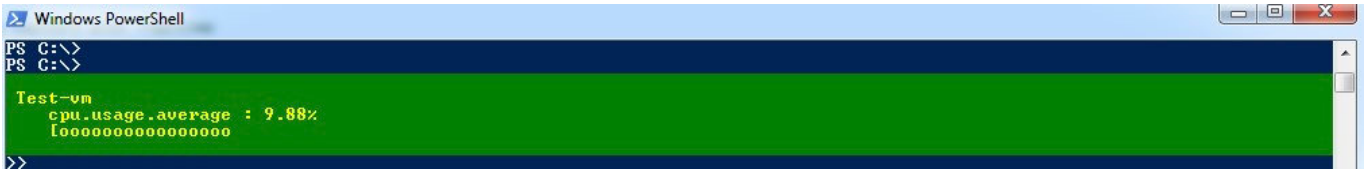
- Retrieve the latest CPU usage counter

```
while ($true) {

  $counter = get-stat -Entity $VM -Cpu -Stat cpu.usage.average -
Realtime -MaxSamples 1 | where unit -eq "%"

  sleep 20
}
```

- Create the progress bar using the values obtained previously

```
 while ($true) {

    $counter = get-stat -Entity $VM -Stat cpu.usage.average -
Realtime -MaxSamples 1 | where unit -eq "%"

    Write-Progress -Activity $VM.name -status "$($counter.metricId)
: $($counter.value)%" -PercentComplete $counter.value

    sleep 20
}
```

- Run the code and enjoy the view. It only refreshes every 20 seconds but it's the best you can do



MONITOR MULTIPLE COUNTERS

We can also add other counters to the progress bar stacked under the VM name. We need to make a few changes to the code. And to make things more interesting we will make a function out of it.

Progress bars can have an ID and a parent ID. You can stack multiple progress bars under another one by specifying a Parent ID. In the example below we added the memory usage that we stacked with CPU usage under the name of the VM.

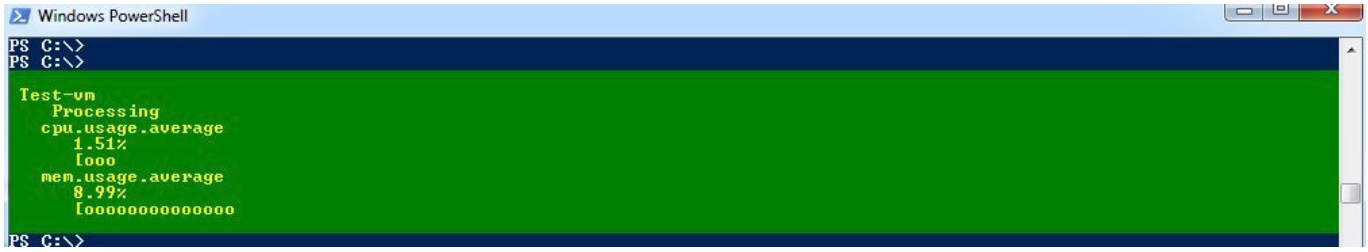
Below is the modified code with added comments. Notice the ID and *ParentID* parameters.

```
Function Invoke-VMMonitor {

param(
    [parameter(position=0,Mandatory=$True)]
    [VMware.VimAutomation.ViCore.types.V1.Inventory.VirtualMachine]
    $VM
)
```

```
while ($true) {
    # Retrieve CPU stat
    $counterCPU = get-stat -Entity $VM -Stat cpu.usage.average -
Realtime -MaxSamples 1 | where unit -eq "%"
    # Retrieve memory stat
    $counterMEM = get-stat -Entity $VM -Stat mem.usage.average -
Realtime -MaxSamples 1 | where unit -eq "%"

    # Create VM progress bar
    Write-Progress -Activity "Test-vm" -Id 1
    # Create CPU usage bar
    Write-Progress -Activity $counterCPU.metricId -status
"$($counterCPU.value)%" -PercentComplete $counterCPU.value -ParentId
1 -id 2
    # Create Memory usage bar
    Write-Progress -Activity $counterMEM.metricId -status
"$($counterMEM.value)%" -PercentComplete $counterMEM.value -ParentId
1 -id 3

    # Wait interval
    sleep 20
    }
}
```

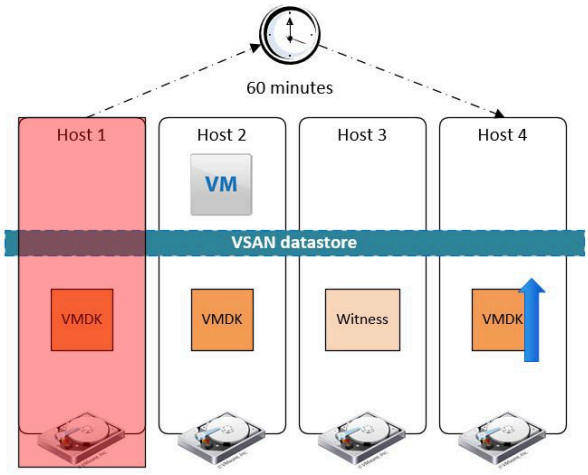
Because it is a function, it is now much easier to run.

```
Invoke-VMMonitor (Get-VM test-vm)
```



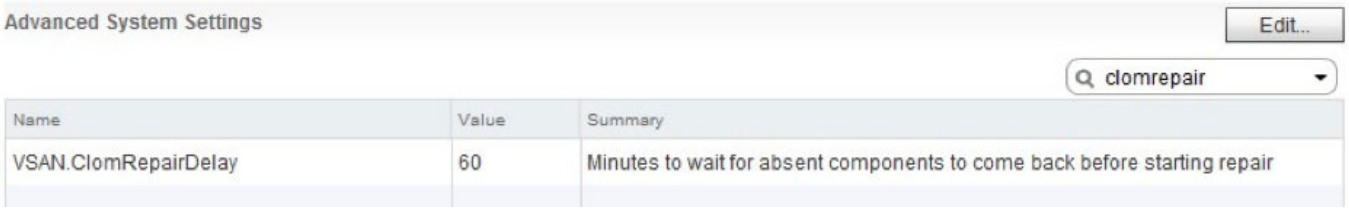
CHANGE THE VSAN DEFAULT REBUILD DELAY

VMware VSAN virtualizes storage across multiple hosts using their local disks. Virtual machines are made of objects that are stripped or mirrored across multiple hosts to achieve redundancy. When a host goes down for planned maintenance or following a failure, the VMs that have objects stored on this host become non-compliant and a 60 minutes countdown is triggered. If the countdown reaches 0, a rebuild of these objects is initiated to restore the compliance with their storage policies.



These rebuilds can put a lot of stress on the disks, reducing their life expectancy and adding CPU overhead during the processing. To avoid such an event occurring during a planned maintenance, it is usually a good idea to increase the duration of the countdown. In the next example, we are going to demonstrate how to do this using PowerCLI to make it quicker and easier than with the UI.

This setting is configured through an advanced setting called *VSAN.ClomRepairDelay*.



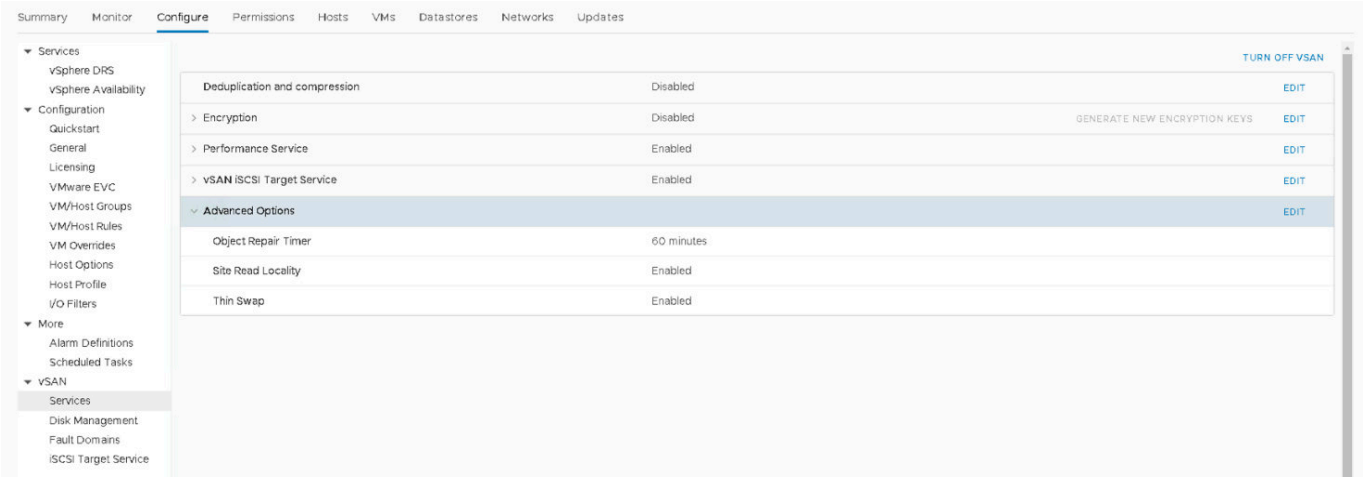
- To change an advanced setting, we need to retrieve the *AdvancedSettings* object of the host to modify one of its properties:

```
$adv = Get-VMHost ESX-Host-1 | Get-AdvancedSetting -Name "VSAN.ClomRepairDelay"
Name                    Value                Type                 Description
----                    -----                ----                 -----------
VSAN.ClomRepairDelay 60                      VMHost
```

- Then we can modify the setting by using the Set command. Some operations like this one require a confirmation in order to proceed. You can bypass this interaction by adding "*-Confirm=$false*" to the command.

```
Set-AdvancedSetting -Value 180 -AdvancedSetting $adv
Perform operation?
Modifying advanced setting 'VSAN.ClomRepairDelay'.
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default
is "Y"): y
Name                 Value                Type                 Description
----                 -----                ----                 -----------
VSAN.ClomRepairDelay 180                  VMHost
```

Note that in VSAN 6.7U1 it is possible to change this setting within the VSAN configuration pane in the advanced options. It is specified that you should wait for at least 180 seconds before proceeding with any maintenance task however after changing this setting.

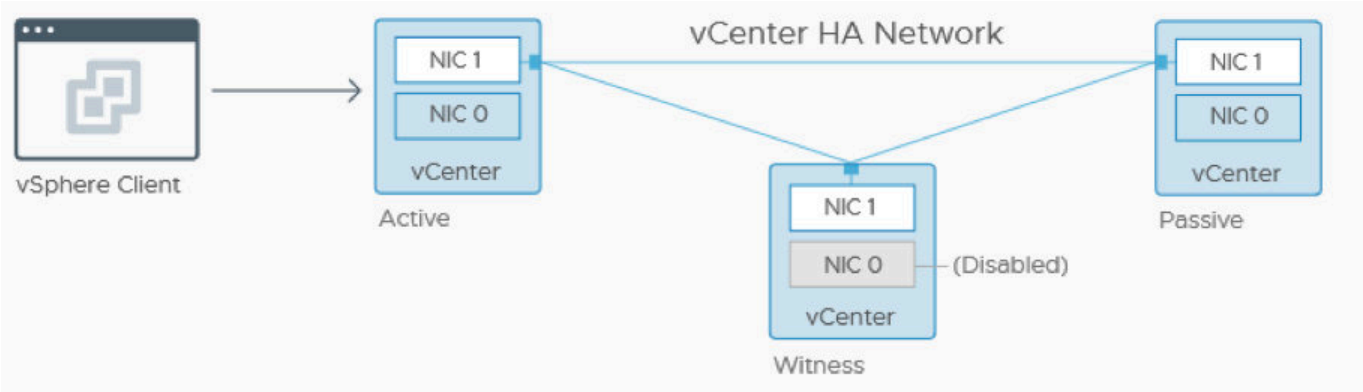


VCENTER HIGH AVAILABILITY (VCHA)

In version 6.5, VMware included a neat new feature that enables the ability to configure high availability (VCHA) at the vCenter level. Much like your average firewall clusters, there is an active node and a passive node as well as a witness, that communicates over a private network.

In order to set up VCHA you have to clone the vCenter appliance to the passive and witness nodes while in VCHA configuration mode. The postgres database is then replicated to the passive node while the witness acts as a tie breaker and doesn't hold any data.

Note: With VCHA, it is possible to size down the witness VM to 1 CPU and 1GB of RAM most of time.



Note that in this section, we won't describe how to set up VCHA with PowerCLI. It is possible but it's something you will probably only do a few times now and then so there is little value in automating it. Therefore, it's preferable to retain more control by doing it manually and focusing your automation efforts on more repetitive tasks that will actually save you time.

VCHA clusters must always be destroyed before performing certain operations such as replacing certificates. If you use this feature in production, always check the relevant documentation before performing an operation on vCenter to see if it is compatible with VCHA.

You may sometimes need to get information about your VCHA cluster, for example, to find out which node is the current active one or to check its status. Unfortunately, PowerCLI offers no cmdlets out of the box to interact with VCHA. Thankfully, the required methods are exposed on the vCenter object which we can leverage to build our own cmdlets. Let's find out how.

THE PROPERTIES AND METHODS

We already worked a little bit with properties and methods in this book, especially when using the *Get-Esxcli* cmdlet. As a quick reminder:

- **Object** - The current entity on which we are working (VM, VMHost, vCenter, etc.).

  In a simple analogy it could be a car for instance.
- **Property** - A single value of a certain type or a collection of objects. In our analogy, the color of the car could be a property of value “red” for instance. It could also contain a property “engine” which will sport a collection of objects with the components that make up the engine.
- **Method** - The actions that can be executed on an object. For our car, a method could be “start engine” on the object “car”.

In the VCHA case, because there are no built-in cmdlets, we will have to use properties and methods to build them ourselves.

The methods and properties in relation to vCenter HA are exposed in the following property on a vCenter object: *$DefaultVIServer.ExtensionData.content.FailoverClusterManager*

GET VCENTER HA STATUS INFORMATION

This function will display the status of the vCenter HA cluster. It contains the health state as well as messages regarding the status of each component.

THE SCRIPT

```
Function Get-VCHAState {

param(
    [VMware.VimAutomation.ViCore.Impl.V1.VIServerImpl]
    $DefaultVIServer = $DefaultVIServer
)

# Ensures that the vCenter is an appliance and not a Windows
install.
if ($DefaultVIServer.ExtensionData.Content.about.OsType -ne "linux-
x64") {
    Write-Warning "VCHA is not available on Windows based vCenter"
    Break
}
```

```
# Obtain the VCHA object that contains the methods and properties.
$failoview = get-view
$defaultviserver.ExtensionData.content.FailoverClusterManager

# Ensure that VCHA is configured. If not a warning displays the
result and breaks the function.
if ((get-view
$defaultviserver.ExtensionData.content.FailoverClusterConfigurator).g
etVchaConfig().state -eq "notConfigured") {
    Write-Warning "VCHA is not configured"
    Break
}

# Get the runtime info in a variable to work with it.
$RuntimeInfo = $failoview.GetVchaClusterHealth().runtimeinfo

# Prepare the powershell object to display.
$Output = @{
    vCenter = $defaultviserver.name
    VCHAstate = $RuntimeInfo.Clusterstate
}

# Fill in the object with each message detected.
foreach ($Message in
$failoview.GetVchaClusterHealth().HealthMessages.Message) {
    $i++
    $Output.Add("Message$($i)",$Message)
}

# Display the object.
[pscustomobject]$Output
}
```

USAGE

The use of this script is as simple as it gets, just run "*Get-VCHAState*" and you will get an output.

If you don't specify the parameter, the current vCenter to which you are connected will be used.

You can specify it in the parameters if you are connected to multiple instances.

```
PS> Get-VCHAState

VCHAstate : healthy
vCenter   : vCenter11
Message2  : Appliance configuration is in sync.
Message4  : Appliance sqlite db is in sync.
Message3  : Appliance state is in sync.
Message1  : PostgreSQL replication mode is Synchronous.
```

In the screenshot above you can see that the VCHA cluster for "*vCenter11*" is healthy and that everything is fine (in sync). This is a simple output straight from the "*GetVchaClusterHealth(). runtimeinfo.nodeinfo*" method. However, if your environment is made up of multiple vCenter instances, it could be interesting to include this in whatever reporting framework you are using to keep track of them.

RETRIEVE VCENTER HA NODE INFORMATION

As mentioned previously, a vCenter HA cluster is made of 3 nodes; active, passive and witness. While the witness node will always remain on the same virtual machine, the VMs that assume the active and passive role can switch. The means the vCenter VM that once was the passive node may be the active one today, so don't just assume VC02 is the passive node if you want to do something on it, you want to double check that information.

You can verify which one is active in the vSphere client, though as usual, why not do it instantly in PowerCLI and skip the browsing through the UI. You can with this script.

THE SCRIPT

```
Function Get-VCHANodes {

param(
    [VMware.VimAutomation.ViCore.Impl.V1.VIServerImpl]
    $DefaultVIServer = $DefaultVIServer
)

# Ensures that the vCenter is an appliance and not a Windows
install.
if ($DefaultVIServer.ExtensionData.Content.about.OsType -ne "linux-
x64") {
    Write-Warning "VCHA is not available on Windows based vCenter"
    Break
}

# Obtain the VCHA object that contains the methods and properties.
$failoview = get-view
```

```
$defaultviserver.ExtensionData.content.FailoverClusterManager

# Ensure that VCHA is configured. If not a warning displays the
result and breaks the function.
if ((get-view
$defaultviserver.ExtensionData.content.FailoverClusterConfigurator).g
etVchaConfig().state -eq "notConfigured") {
    Write-Warning "VCHA is not configured"
    Break
}

# Executes the method to display the runtime info on the VCHA nodes.
$failoview.GetVchaClusterHealth().runtimeinfo.nodeinfo
}
```

USAGE

The use of this function is similar to the previous one. Execute "*Get-VCHANodes*" without a parameter to retrieve information about the VCHA cluster of the currently connected vCenter or specify one in "*-DefaultVIServer*".

```
PS> Get-VCHANodes

NodeState NodeRole NodeIp
--------- -------- ------
up        passive  192.101.0.1
up        active   192.101.0.2
up        witness  192.101.0.3
```

The output will show you the role associated with the IP address on the private network used for vCenter HA. From here you can easily find which VM is active and which one is passive.

GOING FURTHER

As stated before, the PowerCLI exposes the VCHA API on "*$DefaultVIServer.ExtensionData.content.FailoverClusterManager*" so this is where you would start if you wanted to do more with it.

Once you run the *Get-View* cmdlet on this object and store the output in a variable you will be able to see what's inside and what you can do with it. As usual, unless you only work with "*Get*" cmdlets, it is recommended that you practice in a lab environment if possible and never experiment in production.

The following are the most important methods available here:

- **getClusterMode** – Displays the current mode of the VCHA cluster. "enabled", "disabled" or "maintenance"
- **GetVchaClusterHealth** – Displays the last known health of the VCHA cluster
- **initiateFailover** - Commences a failover from the active node to the passive node.

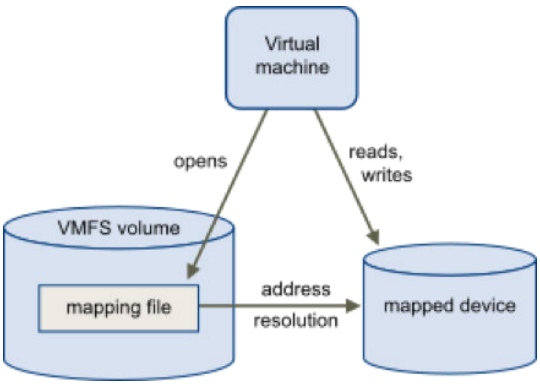
  The cluster mode must be in the "enabled" mode and all cluster members must be present.

  Requires Boolean arguments:
  - **$True**: Failover initiated after the active node flushes its state to passive, no data loss
  - **$False**: Failover initiated immediately, may result in data loss
- **setClusterMode**: Enables changes to the VCHA cluster mode. String argument:
  - **Enabled**: State replication enabled, automatic failover if the active node fails
  - **Disabled**: State replication and automatic failover is disabled
  - **Maintenance**: Automatic failover is disabled while state replication continues

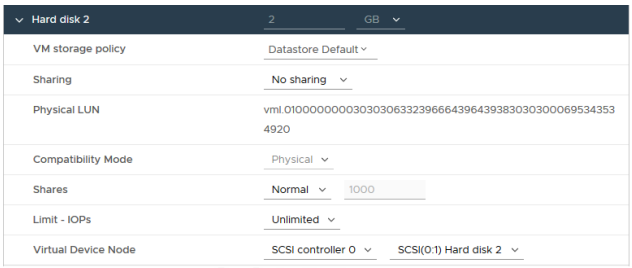
You can also get more methods using "*$defaultviserver.ExtensionData.Content.FailoverClusterConfigurator*" in which you will find the ones associated with the administration of VCHA (configure, create nodes, destroy, deploy, etc.). As mentioned previously, you might have a use case for automation here but it's usually more sensible to perform the initial config and main administration tasks of the cluster manually in the vSphere client.

REPORTING ON RAW DEVICE MAPPING DISKS

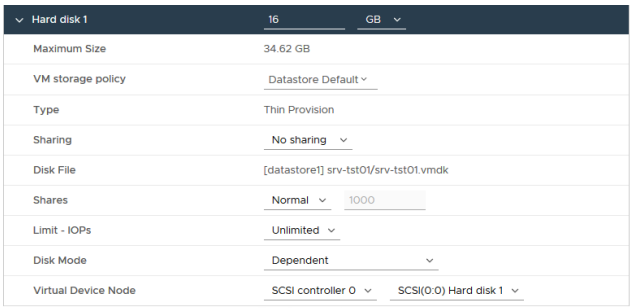
As opposed to traditional virtual disks where the data is stored in a vmdk file, some scenarios require the use of RDM disks (Raw Device Mapping). In this case, there is no -flat.vmdk file that contains the volume's data, instead it acts as a mapping file that redirects the SCSI commands directly to the Logical Unit Number (LUN) storage volume.



A common use case is Microsoft Failover clusters. Because there are multiple Windows server nodes in a failover cluster, the volumes must be locked by only one of them, which would not work with a traditional virtual disk as VMFS (Virtual Machine File System) would intercept the SCSI locking command and discard it as it has its own locking mechanism.



Raw Device Mapping



Traditional disk

It's a rather long introduction to get to the point that RDMs are a special breed of disks which can be tricky to keep track of. In this section we will explain how to quickly get information on the raw disks in the environment. This will prove useful when assembling an inventory or performing any kind of migration as they should be treated differently than regular thin/thick disks.

IDENTIFYING VMS WITH AN RDM DISK

In order to identify the VMs that have an RDM mounted, we can use the "*Get-HardDisk*" cmdlet with the disktype parameter to display the parent object (VM). We also use the "*-Unique*" switch to avoid getting the same VMs for those that have more than one RDM multiple times.

```
get-vm | Get-HardDisk -DiskType RawPhysical,RawVirtual | select -
ExpandProperty parent -Unique
```

OBTAINING INFORMATION ABOUT THE RDMS ON A VM

Finding information about LUNs and RDMs in the vSphere client is cumbersome and takes many clicks to get what you need. Using PowerCLI, you can achieve the same result, and indeed even more, in a lot less time. We are going to build a function that display details about the RDM disk and the SCSI controller it is connected to. Now we are basing this function on VM objects but you could choose to set the parameter on disk objects, LUNs, or whatever else. You can customize the function to fit your environment.

THE SCRIPT

```
Function Get-VMRdm {

Param(
[parameter(position=0,ValueFromPipeline=$True,ValueFromPipelineByPro
pertyname=$True,Mandatory=$True)]
    $VM
)

Process {

    # Process VMs one by one.
    foreach ($V in $VM){

        # Gather list of RDM disks on current VM.
        $Disks = $v | Get-HardDisk -DiskType RawPhysical,RawVirtual
```

```
            # Process RDM one by one.
            foreach ($disk in $Disks) {

                # Identifying the controller.
                $Controller = $V.extensiondata.config.hardware.device |
    Where Key -eq $Disk.ExtensionData.ControllerKey
                $DiskBus =
    "$($Controller.BusNumber):$($Disk.ExtensionData.UnitNumber)"

                # Display the details as a Powershell object.
                [pscustomobject]@{

                    VM = $V.name
                    Name = $Disk.name
                    CapacityGB = $Disk.capacitygb
                    Naa = $Disk.scsicanonicalname
                    Node = "$DiskBus"
                    File = $Disk.filename
                    Persistence = $Disk.Persistence
                    DiskType = $Disk.disktype
                    CompatibilityMode =
    $Disk.ExtensionData.Backing.CompatibilityMode
                    DiskMode = $Disk.ExtensionData.Backing.DiskMode
                    Sharing = $Disk.ExtensionData.Backing.Sharing
                    Controller = $Controller.deviceinfo.summary
                    ControllerSharedBus = $Controller.SharedBus
                }
            }
        }
    }
    }
```

USAGE

This function will display information about the RDM disks connected to the VMs specified in parameter. You can run it against multiple VMs and remember to use the pipe " | ".

In the example below you will see that the VM "Test" has one RDM disk connected. You get to find the NAA (Network Addressing Authority) of its LUN, which type of controller it is, as well as the controller and node ID. These details are particularly useful when working with storage admins to extend, decommission or migrate volumes used as raw disks.

Again, if you have a PowerCLI based reporting framework in place in your environment, you may be interested in adding this to your inventory to get a holistic view of all your RDMs.

```
PS> get-vm test | Get-VMRdm


VM                 : test
Name               : Hard disk 2
CapacityGB         : 2
Naa                : naa.6589cfc000000e853161bb352d4891fc
Node               : 0:1
File               : [datastore1] test/test_1.vmdk
Persistence        : IndependentPersistent
DiskType           : RawPhysical
CompatibilityMode  : physicalMode
StorageFormat      : N/A
DiskMode           : independent_persistent
Sharing            : sharingNone
Controller         : IDE 0
ControllerSharedBus :
```

BACKUP VSPHERE HOST CONFIGURATION

One thing that many admins aren't aware of is that you can back up the configuration of your vSphere hosts. This won't apply to everyone as some prefer to ensure configuration compliance with the use of Host Profiles but others who have standard licenses for instance may get some peace of mind knowing their host configurations are backed up.

If you are in the latter category, VMware thought about you with the "*Get/Set-VMHostFirmware*" cmdlets. Although this may appear a bit confusing at first with the *Firmware* keyword, this will enable you export a backup of your ESXi host configuration to a file.

BACKING UP A VSPHERE HOST CONFIGURATION TO FILE

In the example below we trigger a backup of a host and store the file in the current folder. That's actually all there is to it - it's a very simple process! The file is created and named after the host in .tgz format.

THE CMDLET

```
PS> Get-VMHostFirmware -VMHost 192.168.10.11 -BackupConfiguration -
DestinationPath ./

Host            Data
----            ----
192.168.10.11   C:\Users\xavier\configBundle-192.168.10.11.tgz
```

WRAPPING IT INTO A SCHEDULED TASK

In order to automate the process, we can now wrap this cmdlet into a scheduled task that will keep the last x files tidy in a backup location. Now there are plenty of ways to go about doing this so we are merely suggesting the following one.

The script is formatted as a function so it can be executed against several vCenter servers.

However, it could be turned into a regular script if you only have one vCenter for instance.

To use it you will need to specify a destination where to put the backup files (-BackupLocation), the number of files to keep (-FileRotation) and the name or IP of the vCenter server.

> **Note:** No credentials are specified in this script, which means the user that will run it (scheduled task) needs to have the correct permissions in vCenter

THE SCRIPT

```
Function Backup-ESXiConfig {

param (

    [ValidateNotNullOrEmpty()]
    [string]
    $BackupLocation,

    [ValidateNotNullOrEmpty()]
    [int]
```

```
    $FileRotation,

    [ValidateNotNullOrEmpty()]
    [string]
    $vCenterServer
)

    Connect-VIServer -Server $vCenterServer

    TRY {

        # Processing of each host one by one.
        ForEach ($VMHost in (Get-VMHost)) {

            # Full path to the backup folder.
            $ESXiBackupPath = "$BackupLocation\$($VMHost.name)"

            # Create a folder named after the current host if
doesn't exist.
            IF (!(Test-path $ESXiBackupPath)) {MKDIR
$ESXiBackupPath}

            # Remove the oldest backup files.
            WHILE ((Get-ChildItem $ESXiBackupPath).count -gt
$FileRotation) {

                Get-ChildItem $ESXiBackupPath | Sort-Object
lastwritetime | select -First 1 | Remove-Item -Force -Confirm:$false
            }

            # Take the backup of host config to destination folder.
            Get-VMHostFirmware -VMHost $_.name -BackupConfiguration -
DestinationPath $ESXiBackupPath

            # Rename the backup file to include the date of backup.
            Get-ChildItem $ESXiBackupPath | Sort-Object
lastwritetime | select -Last 1 | Rename-Item -NewName "$(get-date -
Format yyyy-MM-dd)_$($_.name).tgz"
        }

    } CATCH {

        Write-Error $_.Exception -ErrorAction Continue
    } Finally {Disconnect-VIServer -Confirm:$false}
}
```

Note that there is a "*Catch*" component that will display the exception. This can be useful for troubleshooting purposes if you redirect the output of the script execution to a log file.

RESTORING A VSPHERE HOST CONFIGURATION

Restoring the host's configuration is just as easy as backing it up. Once the basic configuration of the host is completed (IP address, etc.) and it is placed in maintenance mode, you can restore a backed-up configuration with the "*Set-VMHostFirmware*" cmdlet.

> **Note:** The build number and UUID of the host must match the ones of the host on backup file.

However, you can force to ignore a UUID mismatch with the "*-Force*" switch. Once again ensure that the host is in maintenance mode.

```
Set-VMHostFirmware -VMHost ESXi01 -Restore -SourcePath
c:\bundleToRestore.tgz -HostUser root -HostPassword P@ssw0rd
```

REPORTING ON SNAPSHOTS

Ever since VMware offered this feature - which was a long time ago - snapshots have been infamous for causing confusion among administrators who are not well versed on virtualization. Now you probably heard a colleague or a contractor stating with a certain sense of passion that "Snapshots are not backups and shouldn't be kept for too long!" Well, they are right. If you are serious about protecting your data, always use a proper backup solution such Altaro VM Backup to back up your virtual machines.

A snapshot records a frozen state of a VM at a certain point in time, which it can be reverted to if desired. This could be useful in a number of scenarios for example a botched software update or simply testing some obscure software or configuration that you don't want persisting in the guest system. The problem with snapshots is that they will grow in size if you don't remove them quickly. In which case it can cause

performance issue and create even more problems when deleting it if the datastore doesn't have enough free space. You don't want to leave snapshots laying around your environment which is the main reason they are not substitutes for backup.

In this section, you'll learn how to report on the snapshot situation and make recommendations based on the age and size of the snapshots. We recommend against removing the snapshots automatically in the script as it is always best to deal with them manually.

You can report on snapshots in a HTML report like we have previously described in this book. Here we will send a table in HTML format in an email when certain thresholds are reached.

THE SCRIPT

```
# Configure according to your environment.
$vcenter = "vcsa.lab.priv"
$From = "vcsa@labpriv.com"
$To = "vmadmin@labpriv.com"
$SmtpServer = "smtp.lab.priv"

# Change values according to your preferences.
$WarningSizeMB = 1024
$CriticalSizeMB = 5120
$WarningDays = 7
$CriticalDays = 15

# Prepare an array to store the snapshot records.
$Table = @()

Connect-VIServer -Server $vcenter

# Process snapshots one by one.
foreach ($snap in (Get-VM | Get-Snapshot)) {

    # Apply status "warning" or "critical" according to the age or
size of the snapshot.
    if     ($snap.created -lt (get-date).AddDays(-$CriticalDays) -or
$snap.sizeMB -gt $CriticalSizeMB) {$status = "critical"}
    elseif ($snap.created -lt (get-date).AddDays(-$WarningDays)  -or
$snap.sizeMB -gt $WarningSizeMB)  {$status = "warning"}

    # If the status variable is populated, the snapshot record is
added to the array.
```

```
    if ($status) {
        $Table += $Snap | Select
@{l="Status";e={$status}},@{l="vCenter";e={$defaultVIServer.name}},V
M,Created,@{l="SizeMB";e={[math]::round($_.SizeMB,0)}},Name,Descript
ion,Quiesced,IsCurrent
        Clear-Variable status
    }
}

if ($Table) {

    # Add borders to the table.
    $Style = "<style>table, th, td {border: 1px solid;}</style>"

    # Convert the array into HTML with the date.
    $HtmlBody = $Table | ConvertTo-HTML -Head $Style -PreContent
"<h1>$(get-date)</h1>"

    # Send the HTML table by email.
    Send-MailMessage -Subject "VM snapshots to check" -BodyAsHtml
$HtmlBody -From $From -to $To -SmtpServer $SmtpServer
}
```

Once the script is running and monitoring the snapshots, every time a snapshot is discovered to be too old or too large, an email will be sent with the list.

01/30/2021 10:42:00

Status	vCenter	VM	Created	SizeMB	Name	Description	Quiesced	IsCurrent
critical	192.168.10.15	test	28/11/2020 14:58:20	0	Restore Point 28-11-2020 14:57:32	<RPData PointTime="5249107704953039844" WorkingSnapshotTime="5249107705157306563" PointSize="128" PointType="ECompleteRestorePoint" DescriptorType="Default" />	False	False
warning	192.168.10.15	test	21/01/2021 10:34:31	2	testing snap		False	True
warning	192.168.10.15	vCLS (1)	21/01/2021 10:38:26	1	snap2		False	True

Note: You can change the thresholds in the script. The ones already in place are rather conservative in terms of space. Once again, the user that runs the script (scheduled task) needs to have permissions on the vCenter server.

RETRIEVING USAGE INFORMATION OF A DATASTORE

Monitoring VMFS datastores is the foundation of the vSphere admin's job. Ensuring that there is enough space to work with and that the overprovisioning ratio isn't getting out of hand is paramount to an environment that leverages thin disks - which is most of them. While thin provisioning is very useful to optimizing the use of your storage, it is important to keep an eye on it as all your thin provisioned VMs would end up in IO failure situation if the datastore were to fill up at 100%.

Here is a useful script that provides a little bit extra information than you would otherwise get which will help you make reasoned decisions with regards to VM placements should you not use datastore clusters. It provides a quick and clear overview of the provisioning of the datastore with a percentage, whether it is expandable or not and how many VMs are running on it.

THE SCRIPT

```
Function Get-VMFSDatastore {
param(

[parameter(position=0,ValueFromPipeline=$True,ValueFromPipelineByPro
pertyname=$True)]

[VMware.VimAutomation.ViCore.Types.V1.DatastoreManagement.VmfsDatast
ore[]]
    $Datastore = (Get-Datastore | where type -eq VMFS)
)
Process{
    ForEach ($DS in $Datastore) {
        if ($ds.type -eq "VMFS") {

            # Check if the datastore is expandable.
            if ($ds.Accessible) {
                $hostId = [string]($ds.ExtensionData.Host | where
{$_.mountinfo.Accessible -and $_.mountinfo.Mounted} | select -
ExpandProperty key -First 1)
                $DsHostDsView = get-view $hostId -Property
ConfigManager.DatastoreSystem
                $DsHostDsView = get-view
$DsHostDsView.ConfigManager.DatastoreSystem
                $Expandable =
$DsHostDsView.QueryVmfsDatastoreExpandOptions($ds.id)
```

```
                if ($Expandable.count -eq 0) {$Expandable = $false}
                else {
                    $LunSize =
($Expandable.info.VmfsExtent.end.block -
$Expandable.info.VmfsExtent.start.block) *
$Expandable.info.VmfsExtent.start.blocksize / 1GB
                    $FreeSpaceOnLun = [math]::round($LunSize -
$ds.CapacityGB,2)
                    $Expandable = "+$($FreeSpaceOnLun)GB"
                }
            } else {$Expandable = $false}

            # Process capacity and provisioning data.
            $CapacityGB    =
[Math]::Round(($ds.extensiondata.summary.capacity   / 1GB),2)
            $FreeGB        =
[Math]::Round(($ds.extensiondata.summary.FreeSpace  / 1GB),2)
            $UsedGB        =
[Math]::Round((($ds.extensiondata.summary.capacity  / 1GB) -
($ds.extensiondata.summary.FreeSpace / 1GB)),2)
            $ProvisionedGB =
[Math]::Round((($ds.extensiondata.summary.capacity  / 1GB) -
($ds.extensiondata.summary.FreeSpace / 1GB) +
($ds.extensiondata.summary.Uncommitted / 1GB)),2)
            $ProvisionedPercent = [Math]::Round($ProvisionedGB /
$CapacityGB * 100,1)
            [pscustomobject]@{
                Name          = $ds.name
                Accessible    = $ds.Accessible
                CapacityGB    = $CapacityGB
                FreeSpaceGB   = $FreeGB
                FreeSpace     = "$([math]::round($FreeGB /
$CapacityGB * 100,1))%"
                UsedSpaceGB   = $UsedGB
                ProvisionedGB = $ProvisionedGB
                Provisioned   = "$ProvisionedPercent%"
                NbRunningVMs  = ($ds | Get-VM | where powerstate -eq
Poweredon).count
                Expandable    = $Expandable
                Lun           =
$ds.ExtensionData.info.vmfs.Extent.diskname
            }
        } else {"$($Datastore.name) is not a VMFS datastore"}
    }
}
}
```

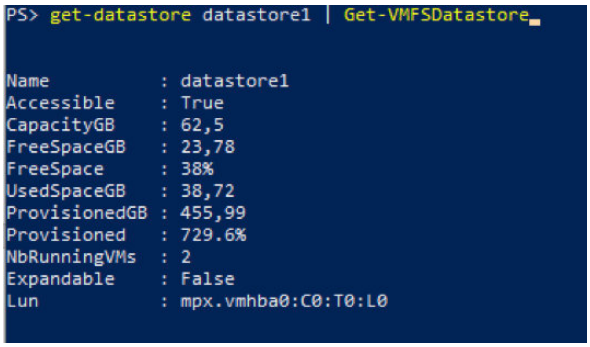
USAGE

If you don't specify a datastore in the $Datastore parameter, all the datastores will be processed.

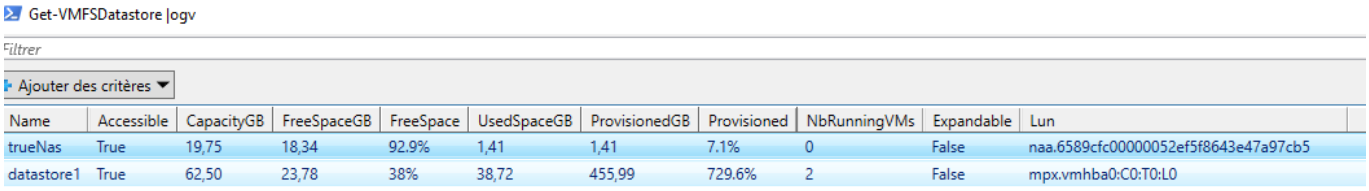
You can use the pipe " | " with this function. Because there are more than 4 fields, the output is displayed as a list. You can get a table by using "| *format-table*" at the end of the command.

```
Get-Datastore dastastore1 | Get-VMFSDatastore
```

As you can see, we get a lot more information here as we now know the provisioning ratio, how many VMs are running on it, if there is unused capacity on the Lun (expandable), etc.



Here is an example of the function with the "*out-gridview*" display format (alias ogv) with no given parameter meaning we process all the VMFS datastores.



CHAPTER 7: RUNNING ESXCLI COMMANDS IN POWERCLI

ESXCLI is a command line tool embedded in ESXi that allows administrators to performs various administrative tasks and manage many aspects of the host with a fairly easy-to-use syntax. ESXCLI commands are particularly useful when troubleshooting situations where traditional means are not available, for instance. These can be run when logged in the interactive shell (DCUI), in SSH sessions, with vCLI, and as we're about to discover, in PowerCLI!

Even though PowerCLI offers more than 600 cmdlets, there is still a large number of tasks that can only be performed with ESXCLI. To fill that gap, VMware provided the powerful *Get-EsxCLI* cmdlet that allows you to run ESXCLI commands from within PowerCLI. It can be a very useful cmdlet when used in conjunction with functions or automated scripts.

It used to be a bit convoluted to use in previous versions as it worked with positional arguments. This means that you had to specify every single argument in the right order, including NULL where no value was required. The -V2 switch was introduced in PowerCLI 6.3R1 and makes *Get-EsxCLI* a lot easier to use.

BROWSING ESXCLI

To start using ESXCLI, run the cmdlet with the -V2 switch, a vSphere host as parameter and store it in a variable that we will name *$EsxCLI*. Then display the content of the variable. As you can see, it is similar to what the esxcli command would output in an SSH session.

```
$EsxCLI = Get-EsxCli -VMHost ESX-Host-1 -V2
```

PowerCLI

```
PS C:\> $EsxCLI = Get-EsxCli -VMHost                    -V2
PS C:\> $EsxCLI

================================
EsxCli:

   Elements:
   ---------
   device
   esxcli
   fcoe
   graphics
   hardware
   iscsi
   network
   nvme
   rdma
   sched
   software
   storage
   system
   vm
   vsan
```

SSH

```
[root@                :~] esxcli
Usage: esxcli [options] {namespace}+ {cmd} [cmd options]

Options:
  --formatter=FORMATTER
                        Override the formatter to use for
  --debug               Enable debug or internal use optio
  --version             Display version information for th
  -?, --help            Display usage information for the

Available Namespaces:
  device                Device manager commands
  esxcli                Commands that operate on the esxcl
                        information.
  fcoe                  VMware FCOE commands.
  graphics              VMware graphics commands.
  hardware              VMKernel hardware properties and c
  iscsi                 VMware iSCSI commands.
  network               Operations that pertain to the mai
                        wide variety of commands to manipul
                        etc) as well as local host IP, DNS
  nvme                  VMware NVMe driver esxcli extensio
  rdma                  Operations that pertain to remote
                        host.
  sched                 VMKernel system properties and comm
  software              Manage the ESXi software image and
  storage               VMware storage commands.
  system                VMKernel system properties and comm
                        system and related system services.
  vm                    A small number of operations that
  vsan                  VMware vSAN commands
```

You can then browse through the $EsxCLI variable just like you would with esxcli.

Except each sub category is a property.

PowerCLI

```
PS C:\> $EsxCLI.software

======================
EsxCliElement: software

   Elements:
   ---------
   acceptance
   profile
   sources
   vib

   Methods:
   --------
   string Help()
```

SSH

```
[root@                :~] esxcli software
Usage: esxcli software {cmd} [cmd options]

Available Namespaces:
  sources               Query depot contents for VIBs and image profiles
  vib                   Install, update, remove, or display individual VIB packages
  acceptance            Retrieve and set the host acceptance level setting
  profile               Display, install, update or validates image profiles
```

RUNNING COMMANDS

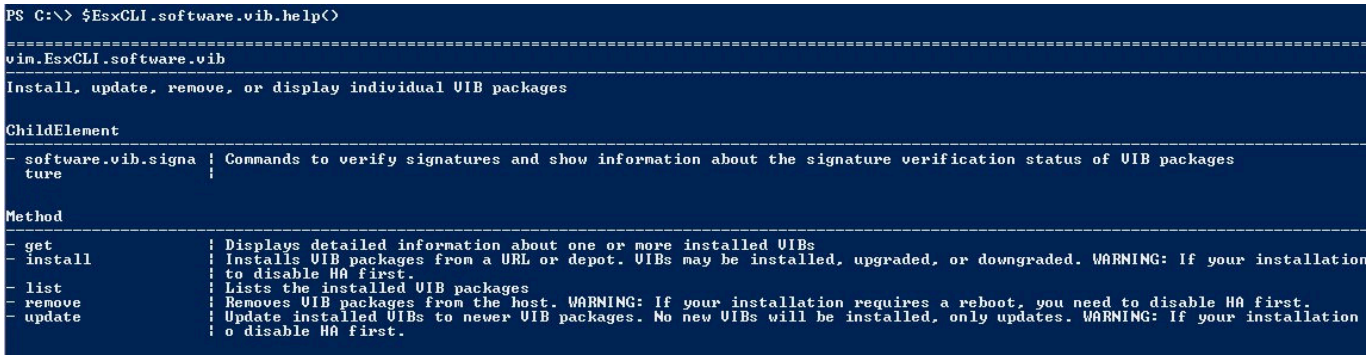
While browsing through the ESXCLI categories is fairly straightforward, running commands differs slightly. Everything is done through the use of methods that we will describe below.

HELP()

This first one is pretty self-explanatory. You can run this method against any ESXCLI object in PowerCLI just like you would in ESXCLI. Take the VIB subcategory, for example, and let's compare it with traditional ESXCLI.

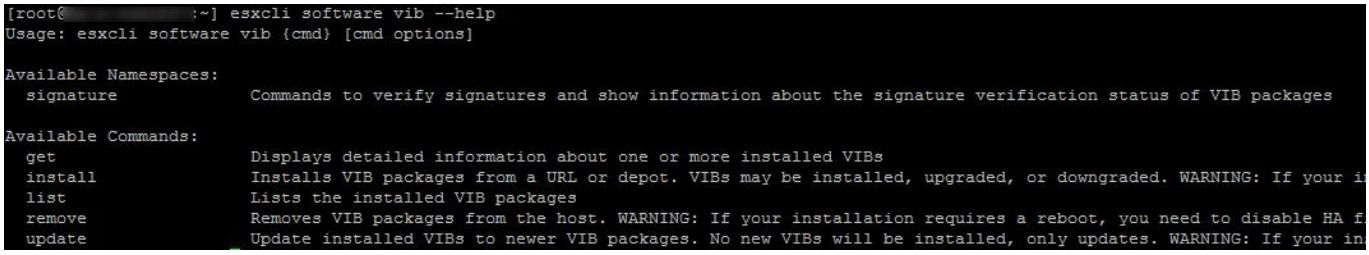
POWERCLI

```
$EsxCLI.software.vib.help()
```



SSH

```
esxcli software vib --help
```



INVOKE()

This method runs the ESXCLI command against the vSphere host. It is comparable to pressing "Enter" in the SSH prompt. Commands that don't require arguments can be run by simply typing "*.Invoke()*" at the end of the command. For instance, to display the installed vib packages:

> **Note:** I added "| *Format-Table*" to compare it with esxcli in SSH, so you will see that the output is very similar.

POWERCLI

```
$EsxCLI.software.vib.list.Invoke() | Format-Table
```

```
PS C:\> $EsxCLI.software.vib.list.Invoke() | Format-Table

AcceptanceLevel CreationDate ID                                                    InstallDate Name                 Status Vendor Version
--------------- ------------ --                                                    ----------- ----                 ------ ------ -------
VMwareCertified 2018-04-03   VMW_bootbank_ata-libata-92_3.00.9.2-16vmw.670.0.0.8169922 2019-03-12 ata-libata-92                VMW    3.00.9.2-16vm...
VMwareCertified 2018-04-03   VMW_bootbank_ata-pata-amd_0.3.10-3vmw.670.0.0.8169922 2019-03-12 ata-pata-amd                    VMW    0.3.10-3vmw.6...
VMwareCertified 2018-04-03   VMW_bootbank_ata-pata-atiixp_0.4.6-4vmw.670.0.0.8169922 2019-03-12 ata-pata-atiixp              VMW    0.4.6-4vmw.67...
VMwareCertified 2018-04-03   VMW_bootbank_ata-pata-cmd64x_0.2.5-3vmw.670.0.0.8169922 2019-03-12 ata-pata-cmd64x             VMW    0.2.5-3vmw.67...
VMwareCertified 2018-04-03   VMW_bootbank_ata-pata-hpt3x2n_0.3.4-3vmw.670.0.0.8169922 2019-03-12 ata-pata-hpt3x2n           VMW    0.3.4-3vmw.67...
VMwareCertified 2018-04-03   VMW_bootbank_ata-pata-pdc2027x_1.0-3vmw.670.0.0.8169922 2019-03-12 ata-pata-pdc2027x          VMW    1.0-3vmw.670....
```

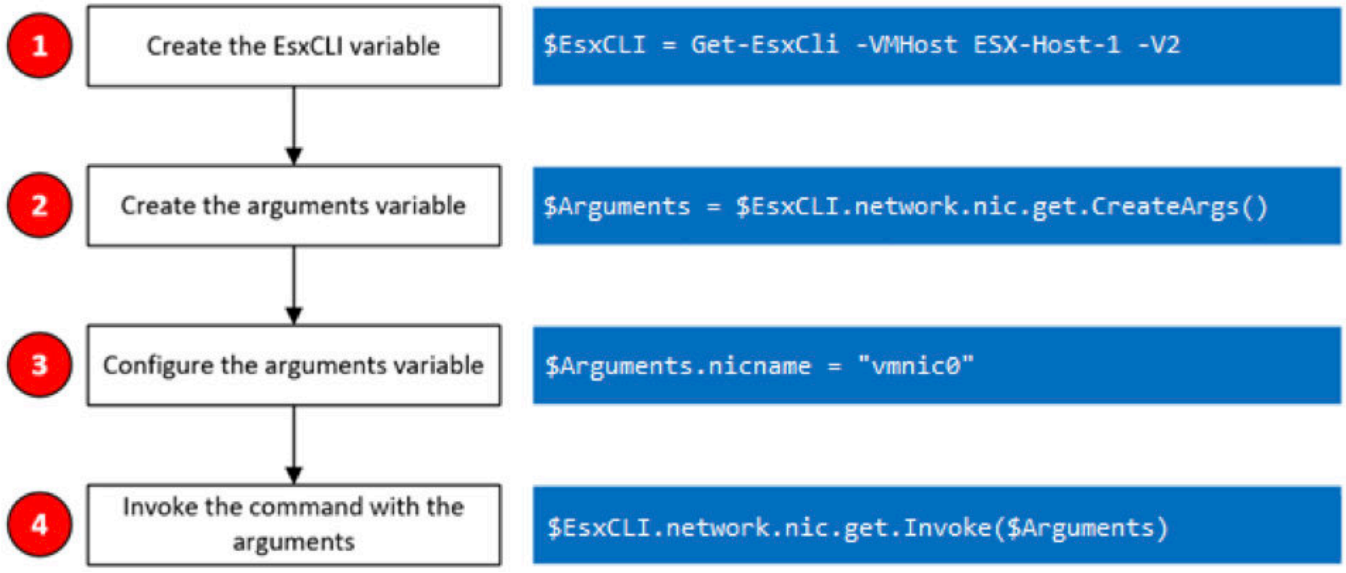
SSH

```
esxcli software vib list
```

```
[root@            :~] esxcli software vib list
Name                           Version                             Vendor  Acceptance Level  Install Date
-----------------------------  ----------------------------------  ------  ----------------  ------------
ata-libata-92                  3.00.9.2-16vmw.670.0.0.8169922      VMW     VMwareCertified   2019-03-12
ata-pata-amd                   0.3.10-3vmw.670.0.0.8169922         VMW     VMwareCertified   2019-03-12
ata-pata-atiixp                0.4.6-4vmw.670.0.0.8169922          VMW     VMwareCertified   2019-03-12
ata-pata-cmd64x                0.2.5-3vmw.670.0.0.8169922          VMW     VMwareCertified   2019-03-12
```

CREATEARGS()

This method is to be used when you need to invoke an ESXCLI command with arguments. This is where the introduction of the *-V2* switch changes things. It is now really easy to create an argument object and populate each of its properties before passing it to the *Invoke()* method. For this example, we are displaying information about an uplink.



POWERCLI

- Create the argument object and display its content

As you can see, only one argument is available to specify the name of the uplink (nicname):

```
PS C:\> $Arguments = $EsxCLI.network.nic.get.CreateArgs()

PS C:\> $Arguments

Name                            Value
----                            -----
nicname                         Unset, ([string])
```

- Populate the arguments object with the uplink name to inspect

```
PS C:\> $Arguments.nicname = "vmnic0"

PS C:\> $Arguments

Name                            Value
----                            -----
nicname                         vmnic0
```

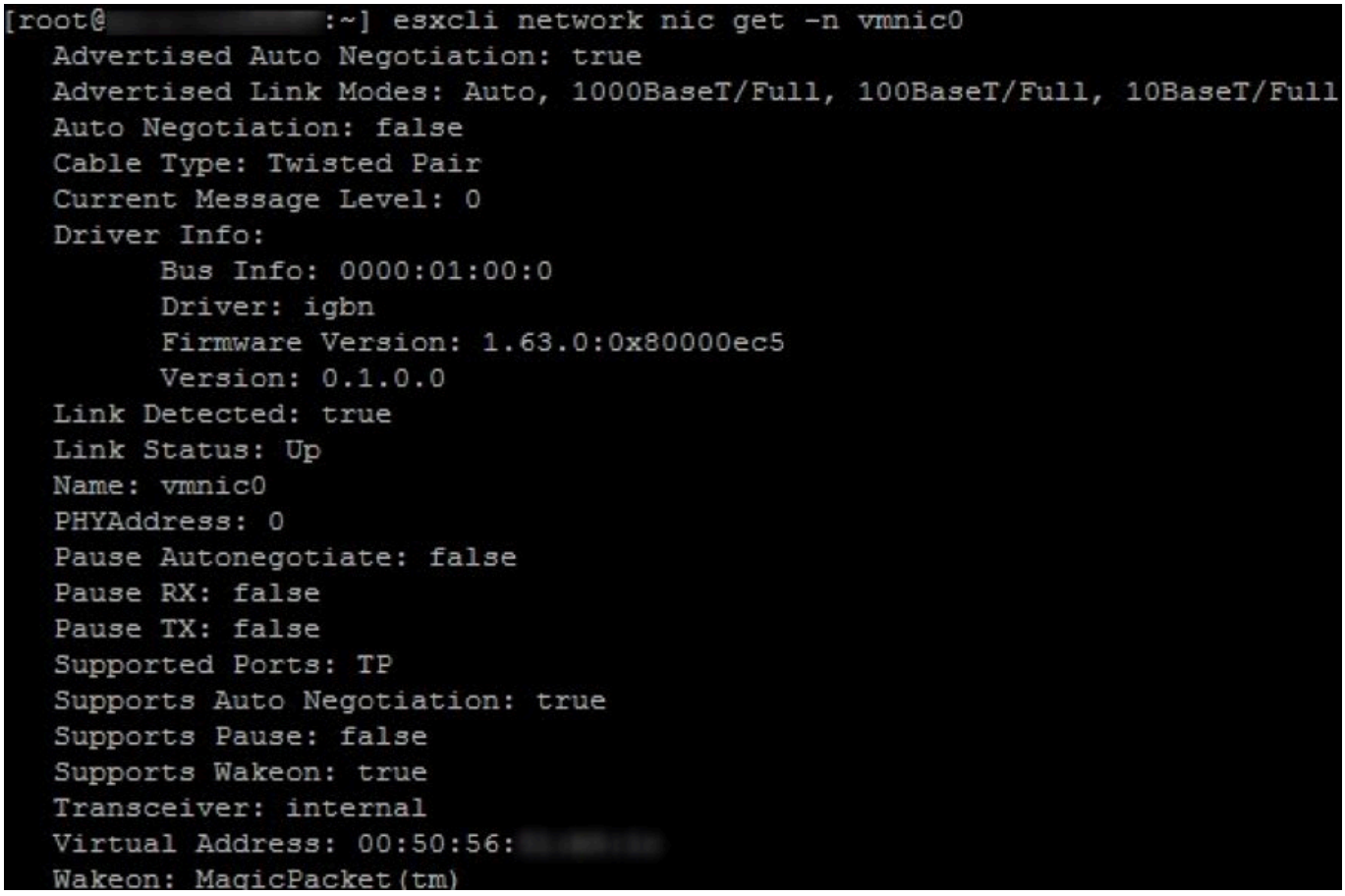
- Invoke the command with the arguments object

```
$EsxCLI.network.nic.get.Invoke($Arguments)
```

```
PS C:\> $EsxCLI.network.nic.get.Invoke($Arguments)


AdvertisedAutoNegotiation : true
AdvertisedLinkModes       : {Auto, 1000BaseT/Full, 100BaseT/Full, 10BaseT/Full}
AutoNegotiation           : false
CableType                 : Twisted Pair
CurrentMessageLevel       : 0
DriverInfo                : VMware.VimAutomation.ViCore.Impl.V1.EsxCli.EsxCliObjectImpl
LinkDetected              : true
LinkStatus                : Up
Name                      : vmnic0
PHYAddress                : 0
PauseAutonegotiate        : false
PauseRX                   : false
PauseTX                   : false
SupportedPorts            : {TP}
SupportsAutoNegotiation   : true
SupportsPause             : false
SupportsWakeon            : true
Transceiver               : internal
VirtualAddress            : 00:50:56:
Wakeon                    : {MagicPacket(tm)}
```

SSH

```
esxcli network nic get -n vmnic0
```



VCENTER AND ESXI API BASED SIMULATOR

One of the biggest hurdles in learning virtualization scripting is the need for resources. You will quickly realize that even the 16GB of memory in your laptop isn't as plentiful as you thought when you bought it. Sure, it's easy enough to a start workstation and spin up a vSphere host but once you add a vCenter appliance it already starts getting toasty. If you want to add more hosts and VMs you better not be too demanding with the vSphere client responsiveness.

vCenter simulator (VCSIM) is a useful tool that has been around for a while now that enables practice without the need for a full-scale environment. It was actually first introduced in the vCenter Server Appliance way back in version 5.1! It was aimed at simulating changes in a vSphere environment.

That way you can simulate a large number of VMs and hosts which you couldn't do in a normal lab due to a lack of resources. Someone going by the alias Nimmis wrapped VCSIM inside a container and released it on Docker hub.

I've decided to cover this container in this book because it is incredibly handy and can get you started with PowerCLI even if you have a 10-year-old netbook with 4GB of memory as all you need is a basic Linux distro to run a docker host. We will quickly show you how to get a docker host running and deploy the VCSIM container on it.

DOCKER ENGINE

- Download and install a linux distribution of your choice (supported by Docker).

  In this example, we are using Debian 10 without a graphical interface

- First you need to update your package index

```
sudo apt-get update
```

- Then install the following dependencies

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

- Add Docker's GPG key. This is optional but then you don't have to worry about the validation of the signatures

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-
key add -
```

- Now we need to install the Docker repository. Note that we set it to stable but you could choose *test* or *nightly*

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/debian \
   $(lsb_release -cs) \
   stable"
```

- Update the package index once again

```
sudo apt-get update
```

- Install the Docker Engine

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Once this is done the docker engine should be ready to run some containers.

DEPLOY THE VCSIM CONTAINER

The great thing about containers is that they are very quick and easy to spin up.

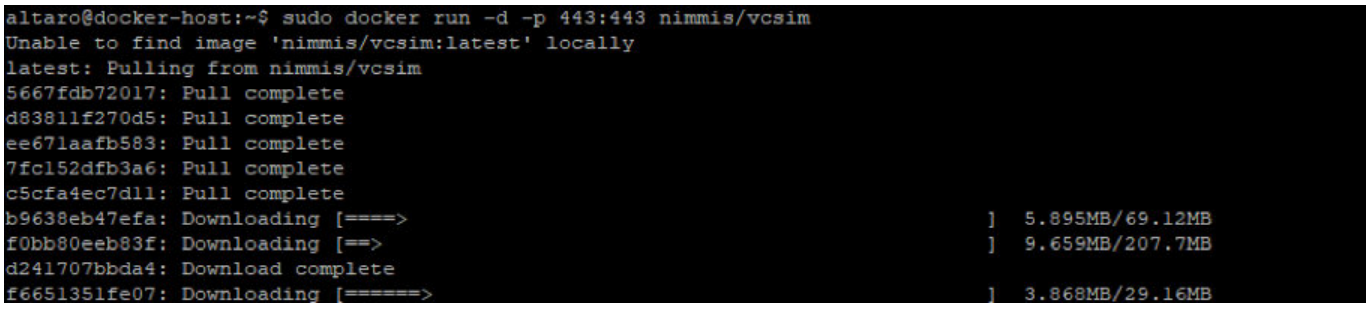
As you will see with the vCenter simulator.

- Run the following command to deploy the VCSIM container and it will execute in “vCenter mode”:

```
sudo docker run -d -p 443:443 nimmis/VCSIM
```

You should get an output such as the following. You will see that the container information

is automatically downloaded from the repository and stored locally on your docker host.



When the operation is completed you can check that the container is running by running the following command.

> **Note:** A random name is attributed to containers in the output (loving_cray in my case).

You will need it to start the container when you reboot the machine.

```
sudo docker container ps
```

```
altaro@docker-host:~$ sudo docker container ps
[sudo] password for altaro:
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS          PORTS                    NAMES
ac8089f63166   nimmis/vcsim   "/docker-entrypoint.…"   15 hours ago    Up 39 seconds   0.0.0.0:443->443/tcp     loving_cray
```

Now you cannot connect to the vSphere web client as there isn't one running but you can connect to it using the API - PowerCLI for instance.

USING POWERCLI WITH VCSIM

Connecting to the simulator is similar to connecting to any other VI instance by using the "*Connect-VIServer*" cmdlet. **Username: u** / **Password: p**

```
Connect-VIServer @IPvCenter -user u -password p
```

Now that you are connected you can start playing with the simulated environment.

Below is a short overview of the environment.

GET-VMHOST

```
PS> Get-VMHost

Name                 ConnectionState PowerState NumCpu CpuUsageMhz CpuTotalMhz   MemoryUsageGB   MemoryTotalGB Version
----                 --------------- ---------- ------ ----------- -----------   -------------   ------------- -------
DC0_H0               Connected       PoweredOn       2          67        7182           1,371           4,000   6.5.0
DC0_C0_H0            Connected       PoweredOn       2          67        7182           1,371           4,000   6.5.0
DC0_C0_H1            Connected       PoweredOn       2          67        7182           1,371           4,000   6.5.0
DC0_C0_H2            Connected       PoweredOn       2          67        7182           1,371           4,000   6.5.0
```

GET-VM

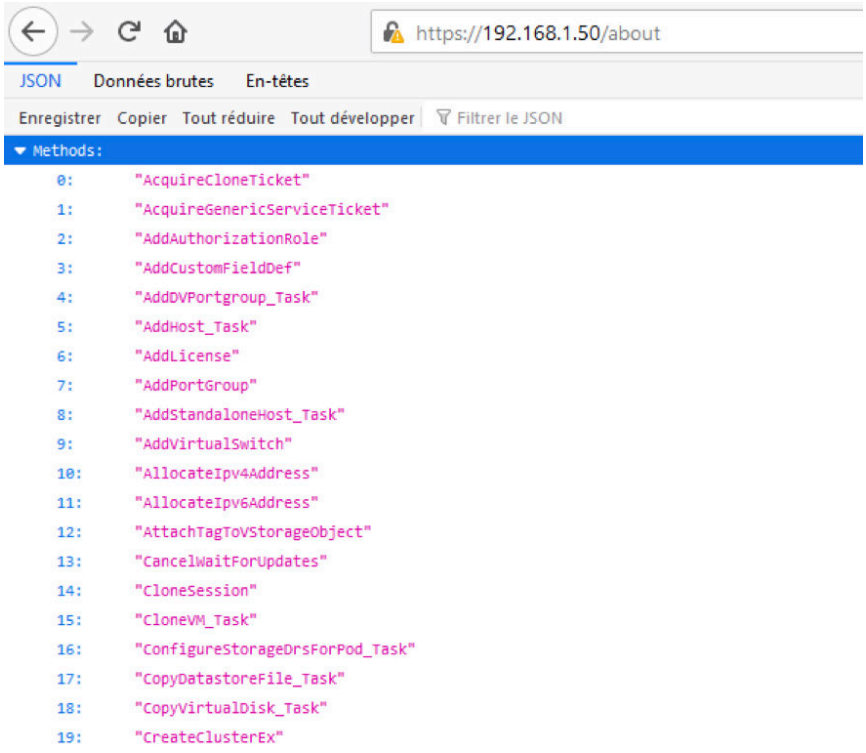
```
PS> Get-VM

Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
DC0_H0_VM0           PoweredOn  1        0,031
DC0_H0_VM1           PoweredOn  1        0,031
DC0_C0_RP0_VM0       PoweredOn  1        0,031
DC0_C0_RP0_VM1       PoweredOn  1        0,031
```

CLONING A VM

```
PS> get-vm *h0*vm0|New-VM -Name "test" -ResourcePool (get-vmhost | select -fi 1)
/VIServer=u@192.168.1.50:443/PartialObject=VirtualMachine-vm-66/

PS> get-vm

Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
DC0_H0_VM0           PoweredOn  1        0,031
DC0_H0_VM1           PoweredOn  1        0,031
test                 PoweredOff 1        0,031
DC0_C0_RP0_VM0       PoweredOn  1        0,031
DC0_C0_RP0_VM1       PoweredOn  1        0,031
```

You will find that many commands don't work as not everything can be simulated in the container.

You can find the list of supported methods if you open a web browser to https://@IPvCenter/about.



When working with VCSIM, you may find oddities here and there compared to a real environment. Meaning if a script of yours fails, it is not necessarily a problem with the script itself, it could very well be that VCSIM that doesn't support the particular action you are performing. This is the drawback of the virtual environment. You will need to double check that what you are doing is actually supposed to work with the simulator if you encounter problems.

For instance, the "*New-VM*" cmdlet's output is a string instead of a VM object which you cannot pipe into another cmdlet like "*Start-VM*".

VCSIM:

```
PS> new-vm -Template test150 -Name "teeetete" -ResourcePool (Get-Cluster)
/VIServer=u@192.168.1.50:443/PartialObject=VirtualMachine-vm-451/
```

VCENTER:

```
PS> new-vm -Template testing -Name "teeetete" -ResourcePool (Get-Cluster)

Name                 PowerState Num CPUs MemoryGB
----                 ---------- -------- --------
teeetete             PoweredOff 1        4,000
```

Now don't take that as a deal breaker as the tool is still incredibly useful, just keep it in mind when working with VCSIM.

CHAPTER 8: CONTINUING YOUR AUTOMATION JOURNEY

VMware went to great lengths in providing the community with a convenient way of interacting with its products using PowerShell with PowerCLI.

In these times of increasing cloud adoption and automation, learning advanced scripting has never been more relevant. PowerCLI is an incredible tool that everyone should have in their toolbox. It unlocks many doors you didn't even know existed before you started using it.

In this book, we barely scratched the surface of what PowerCLI can do. We explained several core concepts and demonstrated several use cases ranging from beginner to medium/advanced usage. While PowerCLI was indeed the main focus, most concepts also apply directly or indirectly to traditional PowerShell and to most well written third-party modules.

After reading this book, the first thing you should do is think about use-cases for PowerCLI that would be useful in your environment. Spend some time experimenting with new scripts and always keep trying to improve them. A good way to progress quickly in PowerShell, in general, is to avoid relying on pre-existing scripts. Instead, take the time to try to write it yourself. The investment will soon pay off as you will learn much faster and will be invaluable when you really need it. If you get stuck somewhere, there is nothing wrong with having a peek at a script you found online to see how he/she did it or ask on the VMTN forums, that's also how you learn.

Once you get comfortable with PowerCLI and end up writing that one awesome script that changed your vSphere admin life, make sure to share it with the community!

I've written a number of articles for the Altaro DOJO on vSphere automation. Be sure to have a browse at those articles and subscribe to the newsletter to be informed when my next article is published!

Thanks for reading!

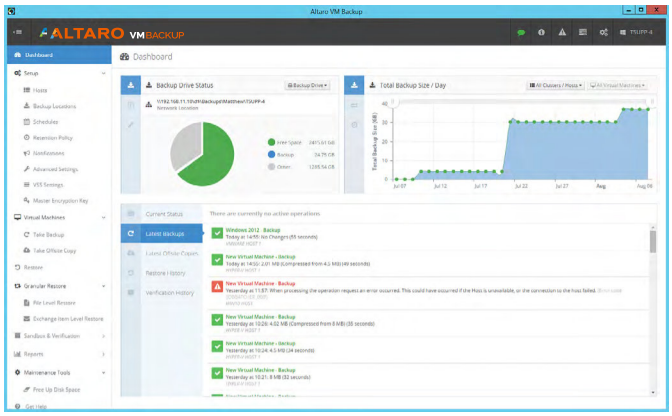
ABOUT AUTHOR

Xavier started his IT career in the early 2010's and evolved from a Windows SysAdmin to virtualization specialist. He now works as a freelancer specializing in VMware technologies for a large customer. Xavier is also a 4 time VMware vExpert and holds VMware VCP certifications. He grew a passion for scripting and automation ever since he started using PowerCLI which offers a fine balance between technicity and creativity.

Xavier also always tries to give back to the online community as much as possible on his blog and the VMware forums by sharing his knowledge and various scripts; as PowerShell says "Help and Get-Help".

ALTARO VM Backup

The virtual machine (VM) Backup and Replication solution built for companies, organizations, MSPs and IT departments.



Fast, fuss-free performance

Hassle-free and fast virtual machine backup in less than 15 minutes.

Best storage savings in the industry

Lower storage requirements by over 65% with our **Augmented Inline Deduplication** technology.

Minimal RPO and RTO

Recovery Point Objective (RPO) and Recovery Time Objective (RTO) through replication and CDP.

Finding a reliable backup and replication solution for Hyper-V and VMware is a real challenge. You risk spending a huge chunk of your budget on software that is bloated with features you don't need, that is hard to manage, and impossible to get support on.

We've created a solid backup and replication solution that does what it says it will; is intuitive, easy to use and well-priced; and most of all, offers 24/7 lightning-fast support as part of the package, with an average call response of less than 30 seconds. Altaro VM Backup is a full-featured, affordable backup and replication solution for Hyper-V and VMware environments.

- More than 50,000 customers
- Price per host or per VM, not per socket
- Award-winning 24/7 support team

spiceworks ★★★★★ 147 Five Star Reviews

STORAGE AWARDS THE STORRIES 2019 WINNER Cloud Product of the Year

INFOSEC AWARDS WINNER CYBER DEFENSE MAGAZINE 2019

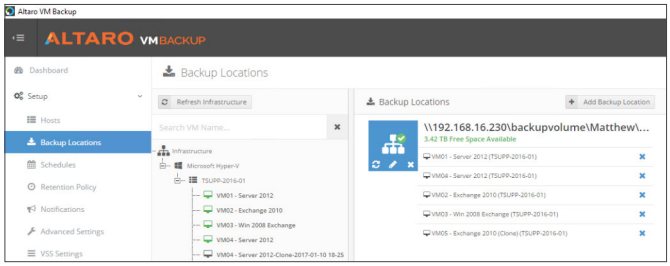
2019 WINNER DCS AWARDS

Virtualization Admin.com Gold Award ★★★★★

START YOUR 30-DAY TRIAL

ALTARO VM Backup

Altaro VM Backup makes it easy to protect your Hyper-V and VMware virtual machines giving you advanced controls and the ability to create and execute a backup strategy you can rely on.



BACKUP

- **Augmented Inline Deduplication -** Reduce backup storage requirements on local/offsite locations, and significantly speed up backups
- **WAN-Optimized Replication -** VMs can be replicated to a remote site continuously, dramatically improving RTO.
- **Continuous Data Protection (CDP) -** back up VMs as frequently as every 5 minutes
- **Concurrency -** Run more efficient backup operations by allowing more VMs to be backed up at the same time
- **Multiple offsite backup locations**
- **Cloud backup to Azure, Amazon S3 and Wasabi**
- **Backup Health Monitor**
- **Flexible backup scheduling**
- **Hot/Live backups**
- **Fast and small backups**
- **Support for MS Hyper-V Clusters (CSV) and VMware vCenter**
- **Compression and military grade encryption**
- **Retention policies**
- **Grandfather-Father-Son archiving (GFS)**

FAST, RELIABLE RESTORES & RECOVERY

- **Boot from Backup** - Instantly boot any VM version from the backup location
- **Instant Business Continuity** - Enable users to be up and running in minimal time should disaster strike by switching to a replicated VM.
- **Restore clones** - Can restore VMs to the same host but with a different name
- **File level restore**
- **Exchange item-level restore** - Restore individual items from backed up VMs
- **Restore VMs to a different host**
- **Sandbox restore & backup verification** - Build and test a recovery plan to ensure you're covered when disaster strikes
- **Granular restore**
- **Fast onepass restores**

FULL CONTROL

- **RESTful API**
- **Integration with Connectwise Automate** (LabTech)
- **Intuitive user interface**

ALTARO VM Backup | CLOUD MANAGEMENT CONSOLE

Altaro's ground-breaking Cloud Management Console (CMC) allows you to manage and monitor all your Altaro VM Backup installations from a single multi-tenant, online console.

BUILT FOR COMPANIES, IT RESELLERS OR MSPS, IT OFFERS:

- **Multi-tenancy:** Enables you to manage and monitor multiple hosts, customers (if you're an IT Reseller or MSP) or sites.
- **Real-time status updates:** View live operation activity and backup results as they occur.
- **Manage your backups/restores:** Start backups and restore VMs, configure backup locations, offsite locations and encryption keys, set default locations, and much more.
- **Access portal anywhere:** There is no need for VPNs to be on-site.
- **Access to Altaro VM Backup for MSPs program:** IT Resellers can manage their Altaro MSP subscription invoicing, billing, etc., and even back up any physical servers.

START YOUR 30-DAY TRIAL

ABOUT ALTARO

Altaro proudly forms part of the Hornetsecurity Group.

Founded in 2009, Altaro has grown rapidly over the years. We develop robust backup solutions to address the core data protection need of businesses and organizations worldwide , as well as the IT resellers, VARs, consultants and managed service providers who serve them.

Out aim is to exceed your data protection needs with reliable, affordable backup solutions backed by an outstanding, personal 24/7 Support team that's determined to help you succeed in protecting your environment.

ALTARO DOJO

The Altaro DOJO is a dedicated training and educational platform for system administrators and IT professionals. It boasts a section dedicated entirely to Vmware content with new content published on a regular basis.

DOJO VMware

Register to the Altaro DOJO now to gain unrestricted access to all Vmware content and stay notified when new content is released - it's free to join!

FOLLOW ALTARO AT:

SHARE THIS EBOOK:

PUBLISHED BY ALTARO SOFTWARE

http://www.altaro.com

WARNING AND DISCLAIMER

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

FEEDBACK INFORMATION

We’d like to hear from you! If you have any comments about how we could improve the quality of this book, please don’t hesitate to contact us by visiting www.altaro.com or sending an email to our Customer Service representative Sam Perry: sam@altarosoftware.com