#PSBlogWeek

Server Management

Table of Contents

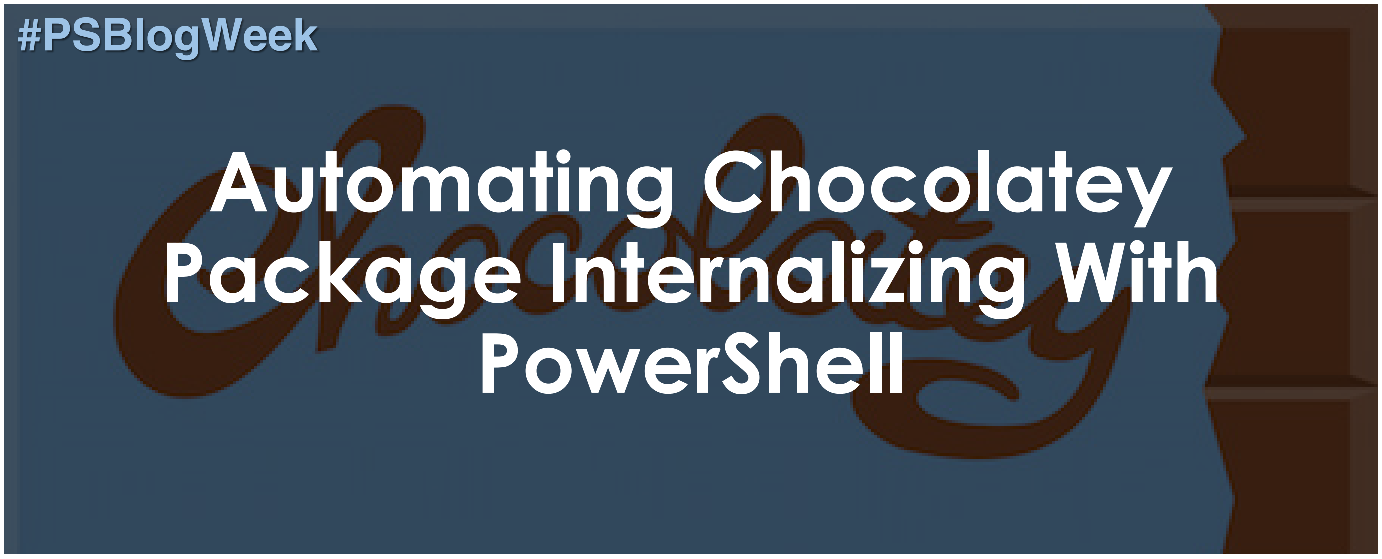
#PSBlogWeek

Intro

#PSBlogWeek is a regular, week-long event where people in the PowerShell community coordinate blogging on a particular topic around Windows PowerShell. The purpose is to pool our collective PowerShell knowledge together over a 5-day period and write about a topic that anyone using PowerShell may benefit from. #PSBlogWeek is a Twitter hashtag, so feel free to stay up-to-date on the topic on Twitter at the #PSBlogWeek hashtag. For more information on #PSBlogWeek or if you'd like to volunteer for future sessions, contact Adam Bertram (@adbertram) on Twitter.

Automating Chocolatey Package Internalizing With PowerShell

Dan Franciscus
Follow on Twitter @dan_franciscus



#PSBlogWeek

Automating Chocolatey Package Internalizing With PowerShell

Many organizations are turning to Chocolatey to manage their Windows packages, and for good reason. Chocolatey allows sysadmins to manage packages completely via the command-line interface (CLI), allowing them to automate the creation, installation, and uninstallation of packages.

If an organization is serious about using Chocolatey to manage its packages, then it really should host its own package repository. Of course, you could use the public Chocolatey repository as your source for packages, but this is a bad idea for a couple of reasons. First, you would have to trust the maintainers of those packages completely, and second, you would need to reach out to the internet to install the packages.

Fortunately, Chocolatey for Business (C4B) allows you to recompile (or internalize) public packages easily, which then enables you to push these packages to your own repository. Internalizing refers to taking remote installers that public packages may call, downloading them, and then embedding them in your NuGet package or another location such as a CIFS share.

When I created my own repository, one of the first tasks I looked at automating was internalizing public packages—especially since I manage hundreds of packages. Doing this has saved me countless hours creating my own packages. The initial internalizing of packages from Chocolatey is actually quite simple, as I will show below, but what happens when a new package version is released? This is where PowerShell comes in handy.

In this article, I will demonstrate how to check for updated Chocolatey packages, test installation, and internalize them automatically. Keep in mind this is done with the Chocolatey Business version, which significantly eases the process.

How Chocolatey internalizing works

Internalizing essentially means Chocolatey will take any remote resources that a public package uses (such as installers), and it will download them locally to create another NuGet package you can use. By default, the resources are relocated into the "tools" directory of the NuGet package. Here I will show a simple example of internalizing the ownCloud client. As you can see in the output, the recompiled package is available in the C:\Example directory:

```
PS C:\Example> choco download owncloud-client --internalize --ignore-dependencies
Chocolatey v0.10.8 Business
Downloading existing package(s) to C:\Example\download
Progress: Downloading owncloud-client 2.3.3.8250... 100%

owncloud-client v2.3.3.8250
Found Install-ChocolateyPackage. Inspecting values for remote resources.
Updating chocolateyInstall.ps1 with local resources.
Recompiling package.

Recompiled package files available at 'C:\Example\download\owncloud-client'
Recompiled nupkg available in 'C:\Example'
```

Now, if we compare the ChocolateyInstall.ps1 file from the public repository to the internalized package we just created, we can see the difference. The internalized package points to the local installer instead of the ownCloud URL, but the checksum is the same:

Public package:

```
$ErrorActionPreference = 'Stop'

$packageArgs = @{
  packageName    = 'owncloud-client'
  fileType       = 'exe'
  softwareName   = 'ownCloud'

  checksum       = '6d347803c779377cf8e66d5235f5a9390b4e328b8d2c7236d5d47f029622c90b'
  checksumType   = 'sha256'
  url            = 'https://download.owncloud.com/desktop/stable/ownCloud-2.3.3.8250-setup.exe'

  silentArgs     = '/S'
  validExitCodes = @(0)
```

Internalized package:

```
$ErrorActionPreference = 'Stop'

$packageArgs = @{
  packageName    = 'owncloud-client'
  fileType       = 'exe'
  softwareName   = 'ownCloud'

  checksum       = '6d347803c779377cf8e66d5235f5a9390b4e328b8d2c7236d5d47f029622c90b'
  checksumType   = 'sha256'
  url            = "$(Split-Path -parent $MyInvocation.MyCommand.Definition)\files\ownCloud-2.3.3.

  silentArgs     = '/S'
  validExitCodes = @(0)
}
```

Internalizing multiple Chocolatey packages

Let's say I have a group of packages I want to internalize. In this example, they're Google Chrome, Git, LastPass, and Notepad++. A recent release of the Chocolatey Business version allows you to pass multiple packages with **choco download**:

```
choco push googlechrome.nupkg --source --api-key='yourapikey'
```

Cool! Now we can just push the packages to our hosted repository:

```
choco download googlechrome git lastpass notepadplusplus --internalize
```

Automating the checking and internalizing for new package versions

Now that we have a set of packages internalized, I want to take this a step further and automate internalizing any new version of a package if it is released on the Chocolatey repository, creating a very basic pipeline. There are a few ways to do this, but what I prefer is to use a virtual machine (VM) that will act as a testing machine for checking, installing, and internalizing new public packages. Set at a certain time interval, such as every 4 hours, this VM will run my script, which includes the command **choco outdated**. This command will see if any Chocolatey packages the machine installed are outdated based on the Chocolatey repository. If the VM finds certain packages have updated versions available, it will first internalize them, then install them to itself, and finally push them to our hosted NuGet server. Pretty cool!

To combine all of this, I created a PowerShell function **Add-ChocoInternalizedPackage**, which orchestrates this process. Here I will step through some of the code to show you what is happening. *Please keep in mind this is a work in progress and currently does not work perfectly*.

The parameter $APIKeyPath indicates that you have an encrypted file that has the API key of your internal Chocolatey repository. The first thing the function does is grab this key to use later on in **choco push**:

```
#Convert API Key Path for use in choco push
$PushAPIKey = Get-Content $APIKeyPath | ConvertTo-SecureString
$BSTR = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($PushAPIKey )
$ApiKey = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($BSTR)
```

Here, we run **choco outdated -r** to get any out-of-date packages on our local machine. If any are found, we create a custom PowerShell object. The output of **choco outdated** needs to be parsed and formatted a bit since "|" is the delimiter, so we use the split method and place them into their own properties. Now, $NewPackages can be processed by the rest of the script.

```
    #Get outdated packages
    $OutdatedPackages = (choco outdated -r)
    #If no updated packages are available then exit
    if (!$OutdatedPackages)
    {
        Write-Warning -Message 'No new packages available. Exiting'
        Exit
    }
    else
    {
        $NewPackages = foreach ($NewPackage in $OutdatedPackages)
        {
            [PSCustomObject]@{
                Name = $NewPackage.Split('|')[0]
                CurrentVersion = $NewPackage.Split('|')[1]
                NewVersion = $NewPackage.Split('|')[2]
                Pinned = $NewPackage.Split('|')[3]
            }
        }
```

We loop through the packages that need updating, but skip any packages with the name *.install that is also contained in the $NewPackages.Name object. The reason for this is that many packages such as “git” are actually virtual packages that just call *.install (example git.install). When using **choco download**, all packages including virtual and dependency packages are downloaded to be internalized, so “git.install” will still be internalized when the “git” package is processed. Skipping *.install packages will save us from redundancy. The $DownloadTime variable is used to filter all packages downloaded after this time, as you will see later.

```
foreach ($InstallPackage in $NewPackages)
{
    #Check to see if need to skip *.install packages due to redundancy in virtual package
    if ($InstallPackage.Name -like "*.install" -and $NewPackages.Name -contains ($InstallPackage.
    {
        Write-Warning ($InstallPackage.Name + ' skipping due to existing virtual package')
        Continue
    }
    #Get time to use with choco push
    $DownloadTime = Get-Date
```

This block of code attempts to internalize the package locally and save it to whatever path is in the $WorkingDirectory variable. Notice we check the $LASTEXITCODE variable to ensure that there were no errors in the **choco download** command. If $LASTEXITCODE is anything but zero (success), we add it to the $Failure array and move to the next package.

```
choco download $InstallPackage.Name --internalize --no-progress
if ($LASTEXITCODE -ne 0)
{
    Write-Warning ($InstallPackage.Name + ' internalize failed')
    $Failure.Add($InstallPackage.Name) | Out-Null
    Continue
}
```

If internalization is successful, it attempts to install that package from the local Chocolatey package. And if the package installs correctly, it pushes all internalized packages created after the $DownloadTime variable (which again includes dependency and virtual packages) to your hosted repository with **choco push**, using the value of $RepositoryURL for your feed.

```
           choco upgrade $InstallPackage.Name --source=$WorkingDirectory --no-progress -y
           #If failure detected in output continue to next package
           if ($LASTEXITCODE -ne 0)
           {
               Write-Warning ($InstallPackage.Name + ' install failed')
               $Failure.Add($InstallPackage.Name) | Out-Null
               Continue
           }
           #If no failure detected than push to hosted repository
           else
           {
               #Get package and all dependency package paths for push
               $DownloadedPackages = Get-ChildItem -Path $WorkingDirectory | Where-Object {$_.Ext
               foreach ($DownloadedPackage in $DownloadedPackages)
               {
                   Write-Output ("Pushing " + $DownloadedPackage)
                   choco push $DownloadedPackage --source=$RepositoryURL -k=$ApiKey
                   if ($LASTEXITCODE -ne 0)
                   {
                       Write-Warning -Message "$DownloadedPackage failed push"
                       $Failure.Add((Split-Path -Path $DownloadedPackage -Leaf)) | Out-Null
                   }
                   else
                   {
                      Write-Output "$DownloadedPackage successfully pushed"
                      $Success.Add((Split-Path -Path $DownloadedPackage -Leaf)) | Out-Null
                   }
               }
           }
```

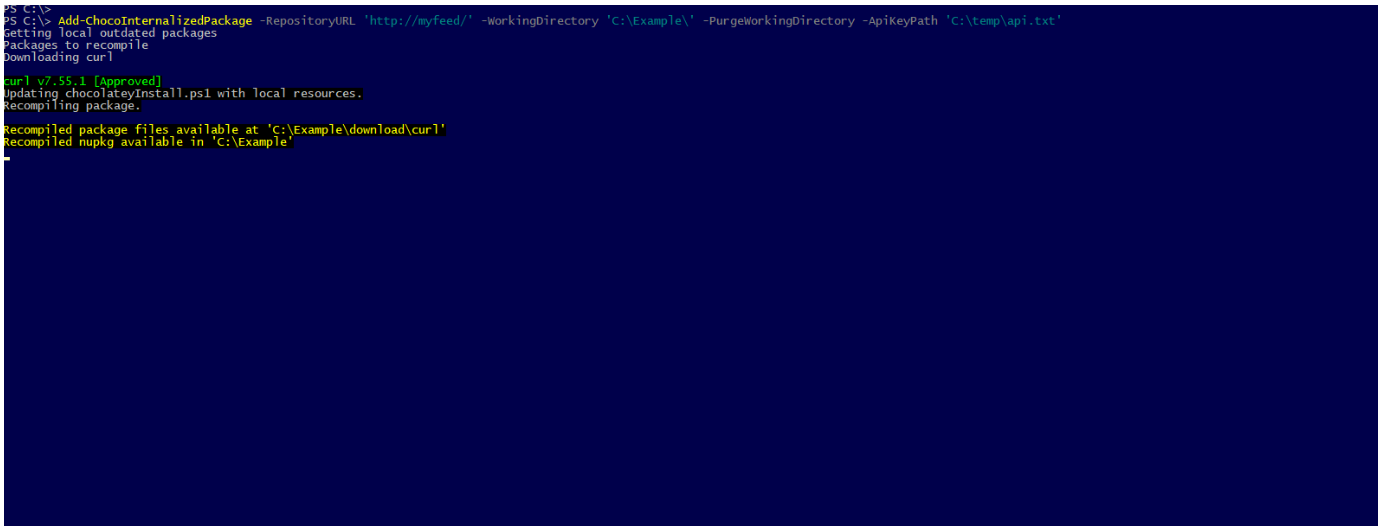
In terms of adding packages to the $Success and $Failure arrays, I chose to use Split-Path and remove the $WorkingDirectory from the value since that is not necessary for the user to see.

Finally, we write the contents of the successful and failed packages to the console. Optionally, you could put this into **Send-MailMessage** and email the results.

```
        #Write successes and failures to console
        Write-Output "Successful packages:"
        $Success
        Write-Output "Failed packages:"
        $Failure
```

See it in action

Here is an example of the function running while it internalizes, installs, and pushes the “curl” package to my repository named “myfeed”. For testing purposes, I am not using https for the internal feed (which is highly recommended). **To increase the size of the gif, please click on it**.



```
PS C:\>
PS C:\> Add-ChocoInternalizedPackage -RepositoryURL 'http://myfeed/' -WorkingDirectory 'C:\Example\' -PurgeWorkingDirectory -ApiKeyPath 'C:\temp\api.txt'
Getting local outdated packages
Packages to recompile
Downloading curl

curl v7.55.1 [Approved]
Updating chocolateyInstall.ps1 with local resources.
Recompiling package.

Recompiled package files available at 'C:\Example\download\curl'
Recompiled nupkg available in 'C:\Example'
```

As in any enterprise environment, you may want to vet the packages you are internalizing, although the Business version of Chocolatey does have some built-in security—such as Virus scanning (against installed A/V or VirusTotal)—not to mention all packages in the public repository go through a rigorous process of approval.

If you care to steal my code, you can do it here on Github .

Logging and Error Handling Best Practices for Automating Windows Update Installs

Darwin Sanoy
Follow on Twitter @DarwinTheorizes

Logging and Error Handling Best Practices for Automating Windows Update Installs

Automation of Windows Updates is generally done by calling wusa.exe against a .MSU file. When wusa.exe experiences a failure the messages are typically obscure and hard to diagnose. The underlying error messages, however, are frequently easy to understand and resolve.

The secret is to have wusa.exe do verbose logging in it's standard exported windows event format and then use the PowerShell event message CMDLets to query that log - automatically, everytime you have an error.

Although focused on applying updates with wusa.exe, this article contains most of my logging best practices for automation coding when calling automated sub-processes.

In This Post

- Toolsmithing Perspective
- Windows Updates Error Trapping is Not Straight Forward
- Examples Of Challenging Messages
- Logging - The Foundation Of Good Exception Handling and Quicker Resolution of Escalations

Toolsmithing Perspective

Everything in this post is helpful even if you are the only one who will ever examine the logs of the automation you build. However, you will hear a persistent theme in this post of how it helps "others". Since I can remember, I have built utility tools that are leveraged by others or supported by a separate support team. In these cases, investments like the ones discussed in this article, have a very high "Team ROI" as well as help to guard the automation developer's time against unnecessary escalation incidents.

Windows Updates Error Trapping is Not Straight Forward

There are seve ral attributes of wusa.exe execution that make attention to logging all the more valuable:

1. wusa.exe critical errors do not generate terminating errors in PowerShell

2. For the same root cause, wusa.exe exit codes are different from logged error codes
3. For the same root cause, logged error codes are in decimal while the ones that might display interactively when running Windows Update GUI are in hex (but at least the same number when converted).

Examples Of Challenging Messages

0x80240017 / 2149842967 / -2145124329 (Wusa.exe exit code) Log: “Windows update could not be installed because of error 2149842967” Interactive “The update is not applicable to your computer” => Painful - first the logged message is not very helpful. Second, you can get this message if you simply do not have the correct prerequisites - if you can’t find the common causes of not meeting prerequisites for the specific update you are applying, then you have to dissect the windows update to read it’s prerequisites section. See “Dissecting Windows Update Packages” below.

0x80070422 / 2147943458 / 1058 (Wusa.exe exit code) - Windows Update Cannot Check For Updates => Usually means the Windows Update Service is stopped - be careful that it is not stopped for a valid reason (e.g. other long running deployment automation is currently underway.)

0x80070002 / 2147942402 / 2 (Wusa.exe exit code) - The system cannot find the file specified. => The .MSU file is not available at the location passed to wusa.exe. Also occurs in other circumstances.

Additional Common Errors

Logging - The Foundation Of Good Exception Handling and Quicker Resolution of Escalations

When coding automation I a huge advocate of verbose logging for everything my automation invokes (as well as having the automation do it’s own logging) - and to a task specific location when logs can be directed to a custom location.

This is for multiple strategic reasons:

1. Verbose sub-process logging is super useful in production environemnts, however, IT IS JAW DROPPING HOW OFTEN VERBOSE LOGGING HELPS ME WHILE DEVELOPING THE AUTOMATION CODE.

2. Once the solution is in production, I get less calls because those who run into problems

with my work can frequently solve the problem on their own when the right logging data is in front of them. That makes me, my team and other teams faster and more productive.
3. When I get calls, I can ask for the logs that I already know exist. That makes me, my team and other teams faster and more productive :)

4. In production (or test or dev) there is no need to attempt to reproduce the problem AFTER logging has been enabled - so if you have a rare transient problem, you get maximum information on every occurence. That makes me, my team and other teams faster and more productive :)

5. I have come across many, many interesting edge cases simply by having verbose logging on for development. That allows me to bake in handling of specific issues when merited or to at least document them and their resolution.

Always Run msu.exe With Verbose Logging

While wusa.exe logging is very useful, there a couple nuances about it that are challenging to navigate - especially if you are starting to use the parameter for the first time.

1. The log output format is “Exported Event Log” - in other words the same type of file as if you exported windows event logs using the event log viewer. Great confusion results if you give it a .log extension because you and others will try to open it in a text editor. A bunch of the productivity benefits I seek are lost if individuals besides myself can’t view the contents without contacting me. To get around this I simply name the log file with the exported event file extension. This causes it to open in the event log viewer if double-clicked and it also cues other people that the file is not a simple text file. I also like to use a file name that communicates a lot of meta data.

The below example tells exactly what update and exactly when the install happened. If people might send me these in email, I’d probably also embed the computer name, because having the wrong log sent to me is a frequent experience that makes us all less productive:

```
$logfilename = "$env:PUBLIC\Logs\ApplyPSH5\PowerShell-Install-W2K12-KB3191565-x64-$(Get
```

2. The command line parameter is fussy. Unlike the other parameters for wusa.exe it:
    1. MUST have a colon after it,
    2. must not have whitespace after the colon,
    3. must be enclosed in quotes if there are spaces in the file name
    4. If you break any of these rules, windows update generally runs without errors, it just does not generate a log.

The above list is an aggregate across multiple versions of Windows - so not all of these sensitivities might be present in a given version of Windows.

Here is an example of what I use to avoid these problems:

```
#Running Directly in CMD or PS1:
wusa.exe W2K12-KB3191565-x64.msu /quiet /norestart /log:"$logfilename"

#Commandline stored in a variable (PS1):
$wusaswitches = "/quiet /norestart /log:`"$logfilename`""
wusa.exe W2K12-KB3191565-x64.msu $wusaswitches
```

Surface The Fact That You Are Doing Detailed Logging

In addition to being a big advocate of logging of called processes, I am a big advocate of verbose logging for my own automation. It is simply mind boggling how many times I find this truncates the root cause determination phase. I will receive a log and immediately notice things like:

1. The wrong version of the automation code has been used.
2. Incorrect parameters were used to call the automation code.
3. Incorrect parameters were used to call the sub-process.
4. The code cannot find something which should be a standard part of every environment it was designed to run in.
5. Exactly what generated “Access Denied”.

This makes me more productive :)

However, another big reason I want to log the fact that I am doing detailed logging on a sub-process is for other professionals to take notice, examine that log and frequently solve the problem on their own - which makes both of us more productive.

Best Practices for Logging The Log Location (and the Calling Command Line)

1. When I do verbose logging of something like wusa.exe I want to surface that fact in the automation log so that others using the work will notice that there is more detailed logging on a specific call available. By ensuring that verbose log location for subprocesses are logged in the top level log, I will help others to follow the chain of available forensic data

in hopes they can solve the problem.

2. I log the sub-process log location all the time, not just if there is an error. This is because there are situations where an error is thrown somewhere else in the script, but these verbose logs reveal the problem or lead to insights that reveal the problem.

3. I also ensure that I Log the exact command line used to call the subprocess. When doing this it is helpful to have stored the command line to run in a variable and use the variable for the actual call as well as the log message. If the same variable is not used, the log can end up being misleading when you update the actual command but forget to find and update the logging lines to reflect that update.

4. In general it is good form to also emit your log messages to the console because if the code runs under any kind of orchestration - that orchestration captures all console output. So for example, if someone is running my code under AWS Cloud Formation - anything emitted in the console is caught up into the general Cloud Formation log - which eliminates one more step in the chain of logs to be followed.

5. It is best to do this logging before calling the sub-process - then if your call causes a catastrophic failure - the log will contain information on the last call made by the automation code (which likely caused the failure if it is reproducible).

Best Practices for Logging Folder Names and Log File Names

Whenever possible I log to system folder (non-user profile) dedicated to the automation task that is running and use a date time serialized string in the name, among other things this helps with:

with troubleshooting discovery of the logs related to the automation (relevant logs in the same folder)

avoids challenges finding logs when run under different user contexts (run under SYSTEM context or a special user profile for a logged on service)

avoids getting removed in automated temp clean up.
can pick a location that every process can write to and every process and logged on technician can read from.

remote access to a standardized “logroot” is easy to setup because it is in a known location and can be shared as read only.

Serialization of Folder and/or File: Ensuring that multiple subsequent runs of the same automation do NOT overwrite each other - overwritten logs complicate troubleshooting as you never know if you are looking at the exact invocation that caused the problem.

Serialization of Folder and/or File: Better date time correlation when searching for logs.

Serialization of Folder: Allows cumulative logs for logging mechanisms that overwrite a previously existing log by default.

Serialization of File: Communicates more meta data when the log is shared with others out of the context of the folder (e.g. email, copy to network).

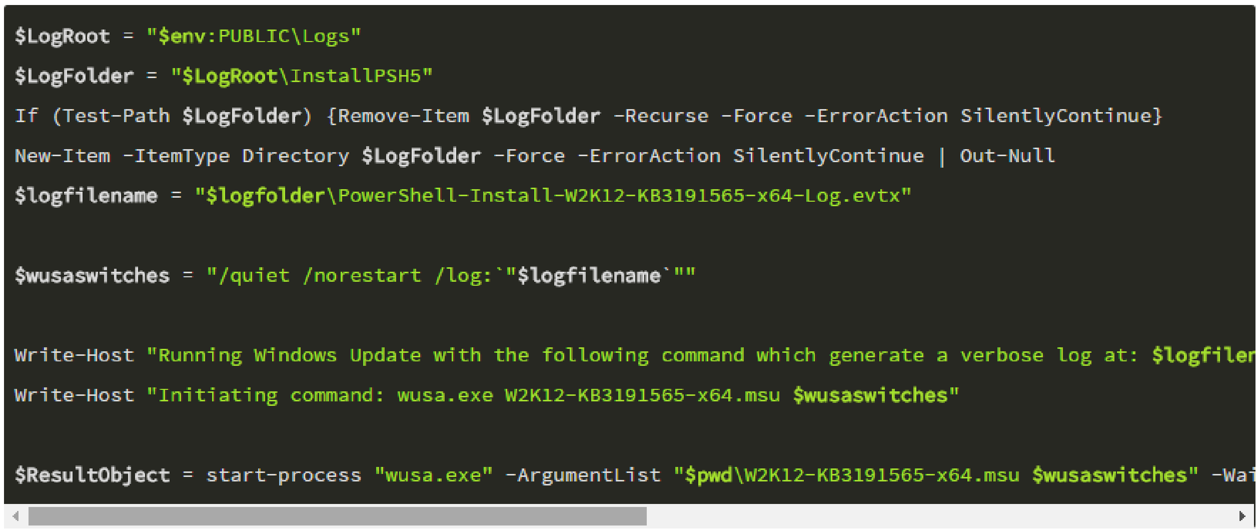
A distinct downside to a custom folder rather than a temp folder is that that you usually have to check for the folder and create it before you start logging to it. A downside to serializing log folder names in and file names in a custom folder that logs can accumulate indefinitely.

env:publicisanenvironmentvariablethatpointstothepublicprofile–whichal
folder can be a great standardized log location, for example $env:public\logs

Example Code

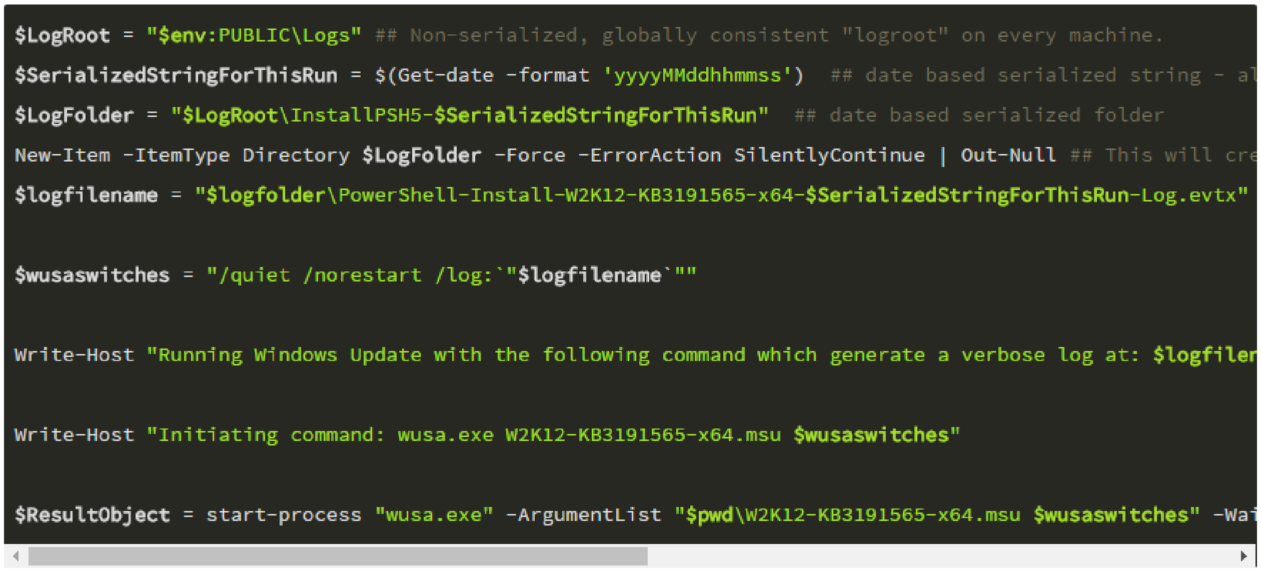
As per the Testable Reference Pattern Manifesto all of the below code has been tested. It is also available in a repository to help avoid problems when copying and pasting from web pages: CloudyWindowsCode

Fully Serialized Example of Log Naming (logs never overwritten - retains historical runs)



```
$LogRoot = "$env:PUBLIC\Logs"
$LogFolder = "$LogRoot\InstallPSH5"
If (Test-Path $LogFolder) {Remove-Item $LogFolder -Recurse -Force -ErrorAction SilentlyContinue}
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue | Out-Null
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-Log.evtx"

$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a verbose log at: $logfiler
Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

$ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64.msu $wusaswitches" -Wai
```

Non-serialized Example of Log Naming, reset logs to clean every-time (does not retain history)



```
$LogRoot = "$env:PUBLIC\Logs" ## Non-serialized, globally consistent "logroot" on every machine.
$SerializedStringForThisRun = $(Get-date -format 'yyyyMMddhhmmss')  ## date based serialized string - al
$LogFolder = "$LogRoot\InstallPSH5-$SerializedStringForThisRun"  ## date based serialized folder
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue | Out-Null ## This will cre
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-$SerializedStringForThisRun-Log.evtx"

$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a verbose log at: $logfiler

Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

$ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64.msu $wusaswitches" -Wai
```

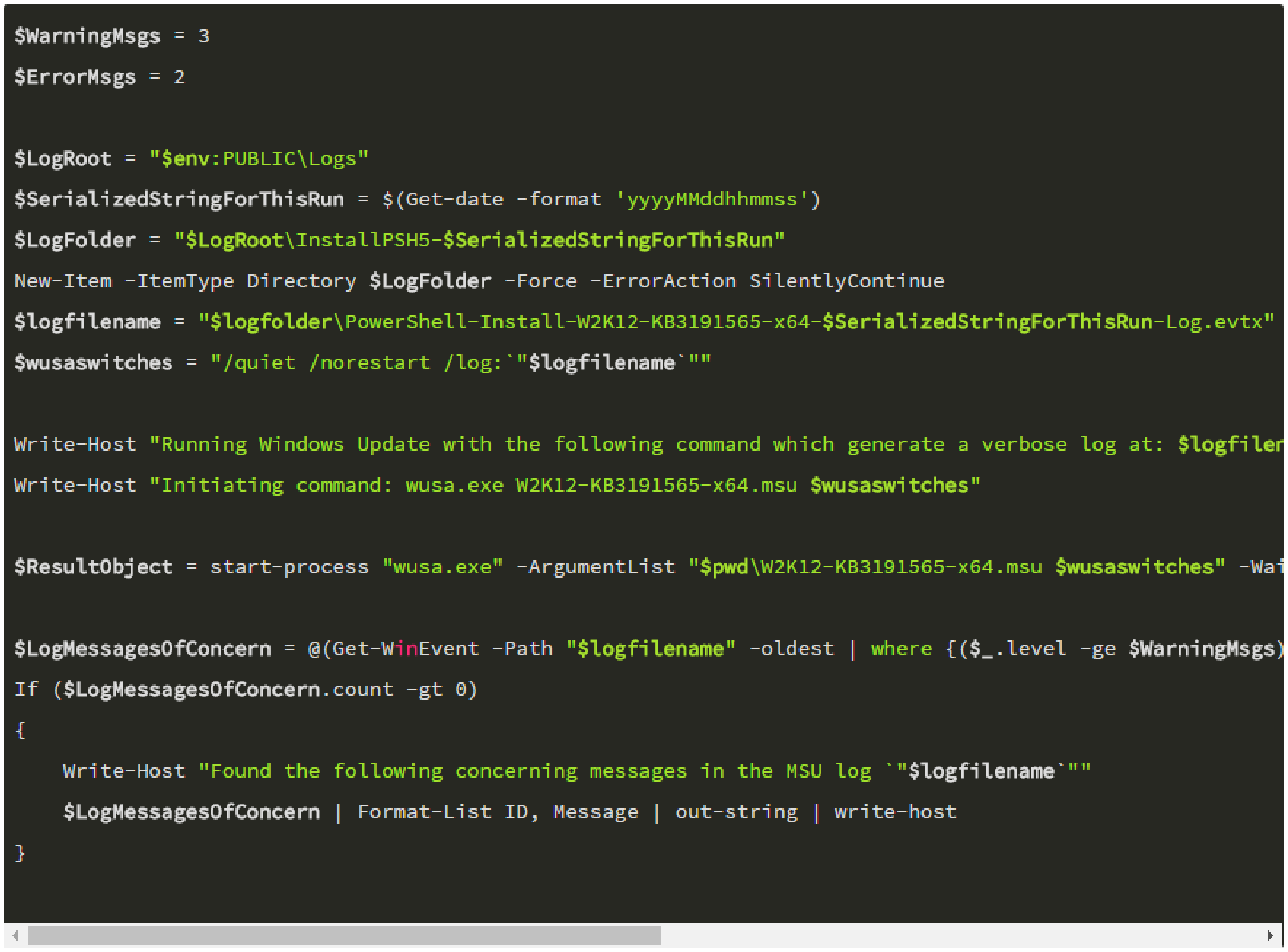
Surface Error Messages (and Warnings If You Wish)

The below snippets show various ways of retrieving and potentially acting upon the messages generated by wusa.exe.

You can creatively mix some of these together - for instance, always re-logging found warnings and errors - but only taking action if ther are actually errors.

Always Show All Messages in Log

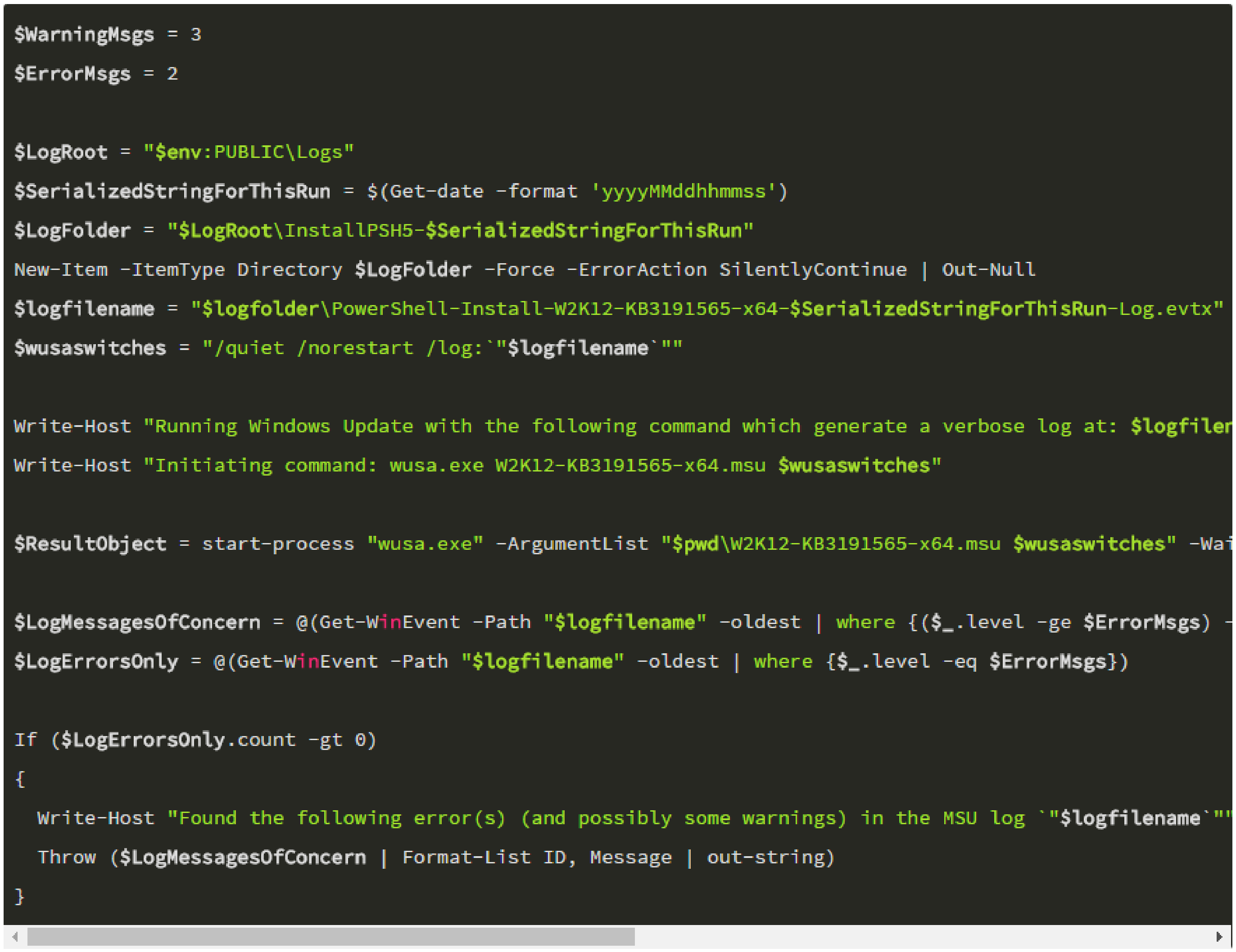
This code always re-logs the detailed warnings and errors from the windows uupdate log - this is a good general practice for debugging related errors as well as helping with messages that are classfied as “Warnings” by windows update, which are essentially errors for your deployment scenario.



```
$WarningMsgs = 3
$ErrorMsgs = 2

$LogRoot = "$env:PUBLIC\Logs"
$SerializedStringForThisRun = $(Get-date -format 'yyyyMMddhhmmss')
$LogFolder = "$LogRoot\InstallPSH5-$SerializedStringForThisRun"
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-$SerializedStringForThisRun-Log.evtx"
$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a verbose log at: $logfiler
Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

$ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64.msu $wusaswitches" -Wai

$LogMessagesOfConcern = @(Get-WinEvent -Path "$logfilename" -oldest | where {($_.level -ge $WarningMsgs)
If ($LogMessagesOfConcern.count -gt 0)
{
    Write-Host "Found the following concerning messages in the MSU log `"$logfilename`""
    $LogMessagesOfConcern | Format-List ID, Message | out-string | write-host
}
```

Using the Existence of an Error in the Verbose Log

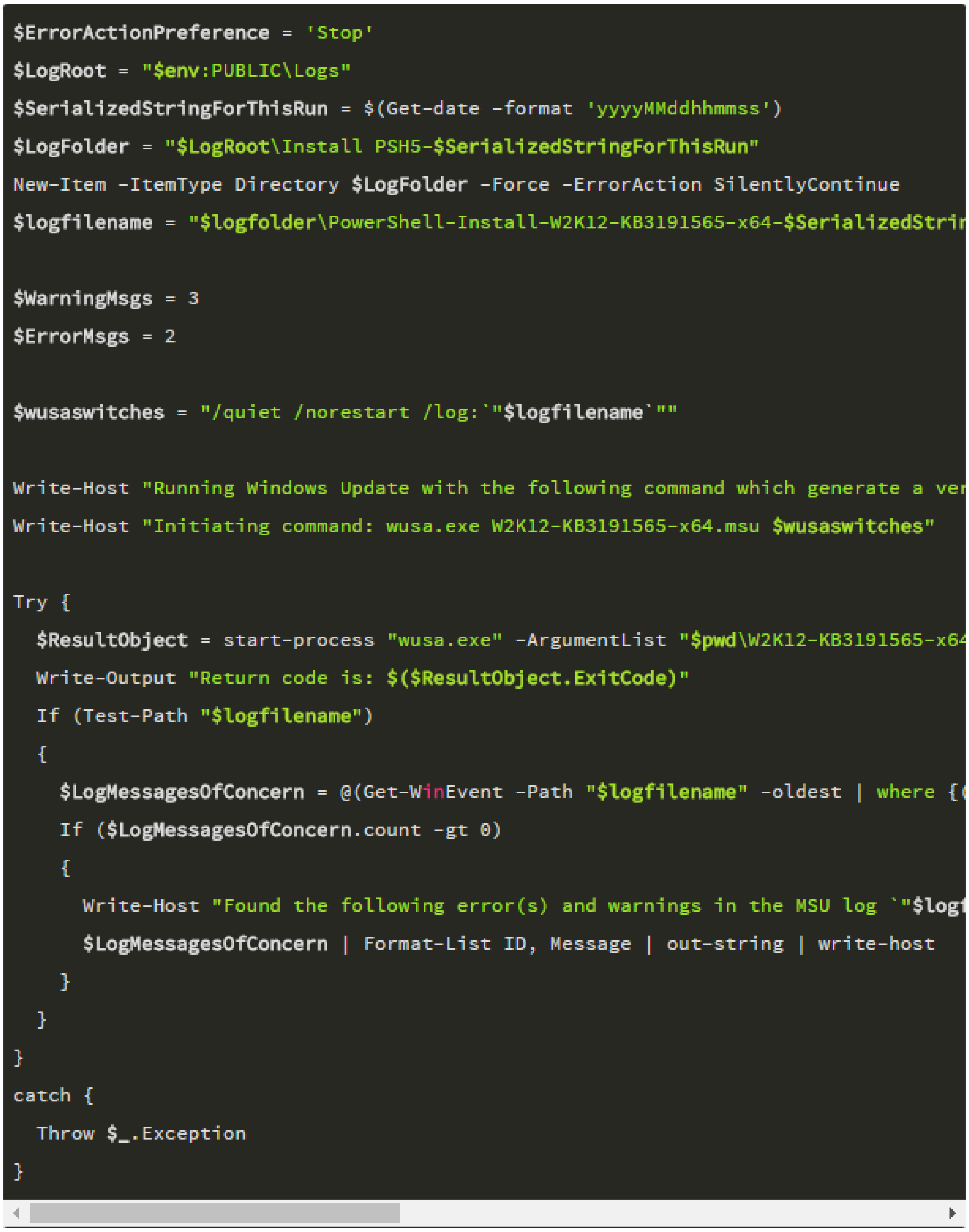
The code relies on the windows update to accurately report errors and then consider that there is a problem only if that is true. A downside to this code is that it will not capture mal-formed wusa.exe command lines or a non-existent .msu.



```
$WarningMsgs = 3
$ErrorMsgs = 2

$LogRoot = "$env:PUBLIC\Logs"
$SerializedStringForThisRun = $(Get-date -format 'yyyyMMddhhmmss')
$LogFolder = "$LogRoot\InstallPSH5-$SerializedStringForThisRun"
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue | Out-Null
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-$SerializedStringForThisRun-Log.evtx"
$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a verbose log at: $logfilen
Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

$ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64.msu $wusaswitches" -Wai

$LogMessagesOfConcern = @(Get-WinEvent -Path "$logfilename" -oldest | where {($_.level -ge $ErrorMsgs) -
$LogErrorsOnly = @(Get-WinEvent -Path "$logfilename" -oldest | where {$_.level -eq $ErrorMsgs})

If ($LogErrorsOnly.count -gt 0)
{
  Write-Host "Found the following error(s) (and possibly some warnings) in the MSU log `"$logfilename`""
  Throw ($LogMessagesOfConcern | Format-List ID, Message | out-string)
}
```

Adding Try / Catch - Just Display Errors and Warnings

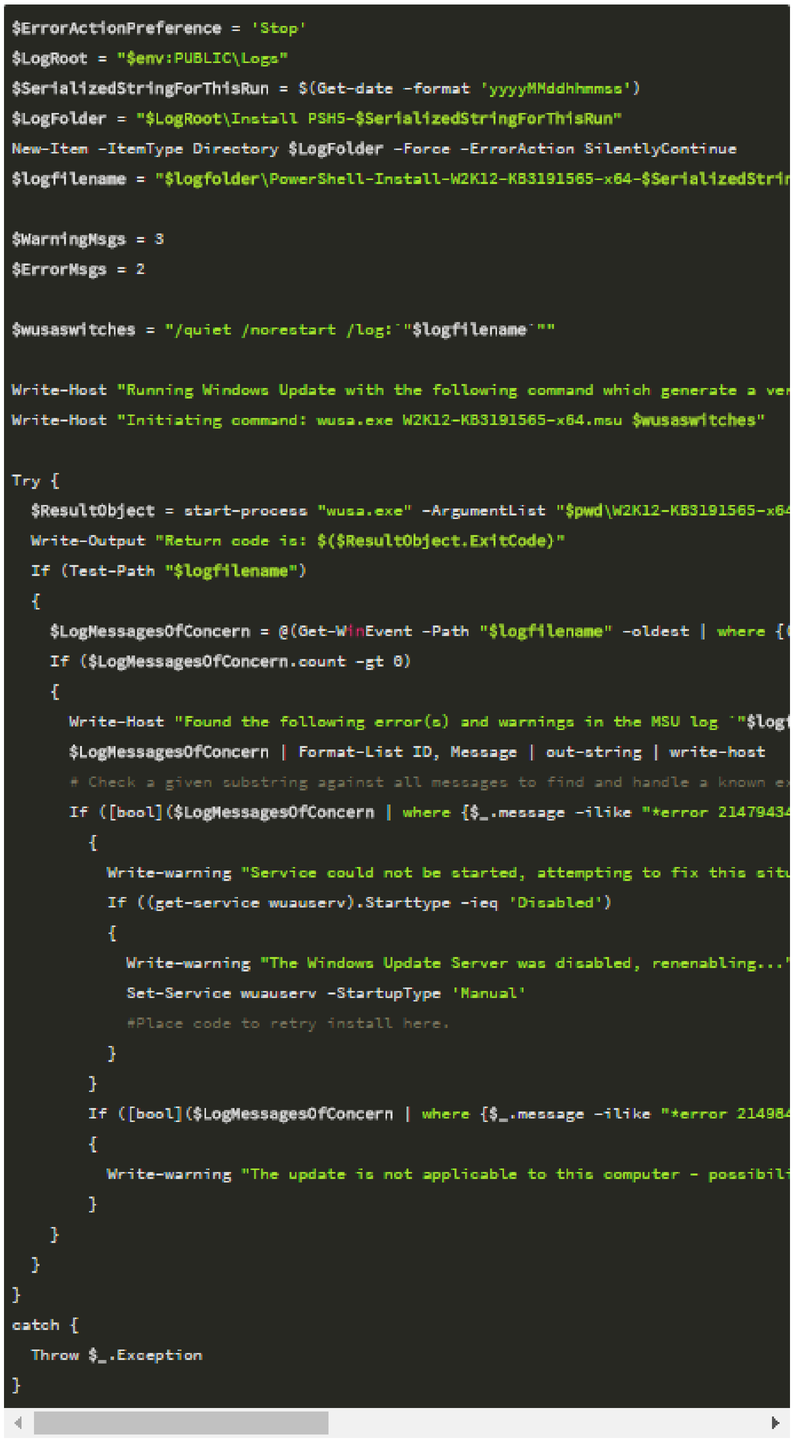
Try/Catch should be used to capture exceptions that happen because wusa.exe cannot be found or started at all.



```
$ErrorActionPreference = 'Stop'
$LogRoot = "$env:PUBLIC\Logs"
$SerializedStringForThisRun = $(Get-date -format 'yyyyMMddhhmmss')
$LogFolder = "$LogRoot\Install PSH5-$SerializedStringForThisRun"
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-$SerializedStrin

$WarningMsgs = 3
$ErrorMsgs = 2

$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a ver
Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

Try {
  $ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64
  Write-Output "Return code is: $($ResultObject.ExitCode)"
  If (Test-Path "$logfilename")
  {
    $LogMessagesOfConcern = @(Get-WinEvent -Path "$logfilename" -oldest | where {(
    If ($LogMessagesOfConcern.count -gt 0)
    {
      Write-Host "Found the following error(s) and warnings in the MSU log `"$logf
      $LogMessagesOfConcern | Format-List ID, Message | out-string | write-host
    }
  }
}
catch {
  Throw $_.Exception
}
```

Acting Upon Specific Messages

The following code shows how to take action depending on the CONTENT of the error in the verbose windows update / wusa.exe log. The action in this case is simply to give a more meaningful error - but if the error is recoverable, you can use the same code block to take action.



```
$ErrorActionPreference = 'Stop'
$LogRoot = "$env:PUBLIC\Logs"
$SerializedStringForThisRun = $(Get-date -format 'yyyyMMddhhmmss')
$LogFolder = "$LogRoot\Install PSH5-$SerializedStringForThisRun"
New-Item -ItemType Directory $LogFolder -Force -ErrorAction SilentlyContinue
$logfilename = "$logfolder\PowerShell-Install-W2K12-KB3191565-x64-$SerializedStrin

$WarningMsgs = 3
$ErrorMsgs = 2

$wusaswitches = "/quiet /norestart /log:`"$logfilename`""

Write-Host "Running Windows Update with the following command which generate a ver
Write-Host "Initiating command: wusa.exe W2K12-KB3191565-x64.msu $wusaswitches"

Try {
  $ResultObject = start-process "wusa.exe" -ArgumentList "$pwd\W2K12-KB3191565-x64
  Write-Output "Return code is: $($ResultObject.ExitCode)"
  If (Test-Path "$logfilename")
  {
    $LogMessagesOfConcern = @(Get-WinEvent -Path "$logfilename" -oldest | where {(
    If ($LogMessagesOfConcern.count -gt 0)
    {
      Write-Host "Found the following error(s) and warnings in the MSU log `"$log
      $LogMessagesOfConcern | Format-List ID, Message | out-string | write-host
      # Check a given substring against all messages to find and handle a known ex
      If ([bool]($LogMessagesOfConcern | where {$_.message -ilike "*error 21479434
        {
          Write-warning "Service could not be started, attempting to fix this situ
          If ((get-service wuauserv).Starttype -ieq 'Disabled')
          {
            Write-warning "The Windows Update Server was disabled, renenabling..."
            Set-Service wuauserv -StartupType 'Manual'
            #Place code to retry install here.
          }
        }
        If ([bool]($LogMessagesOfConcern | where {$_.message -ilike "*error 214984
        {
          Write-warning "The update is not applicable to this computer - possibili
        }
    }
  }
}
catch {
  Throw $_.Exception
}
```

Dissecting Windows Update Packages

This is most frequently done to attempt to understand the prerequisites or OS targeting of a given update in order to understand why it gets the error 0x80240017 / 2149842967 / -2145124329.

```
New-Item $env:PUBLIC\Logs\ApplyPSH5\updatefiles -ItemType Directory
expand W2K12-KB3191565-x64.msu -F:* $env:PUBLIC\Logs\ApplyPSH5\updatefiles
Get-Content $env:PUBLIC\Logs\ApplyPSH5\updatefiles\*.txt
```

This Code In Production

This code is used in the PowerShell for Windows Chocolatey package. It is used to surface better details about why the ugprade failed.

This package has been installed over 1.1 million times - so exposing the underlying errors for others to resolve on their own guards my time and keeps me a lot more productive ;)

Chocolate Package for PowerShell for Windows (see the 'catch' statement near the bottom of this script): https://github.com/DarwinJS/ChocoPackages/blob/master/PowerShell/v5.1/tools/ChocolateyInstall.ps1

Updated Code In This Article

The below code's primary home is on the following repository - where it might be improved upon compared to the below. It is also safer to use the code from the repo rather than copy and paste from this post: https://github.com/DarwinJS/CloudyWindowsAutomationCode

Additional Common Errors

0x80070003 / 2147942403 - The system cannot find the file specified. => The .MSU file is not available at the location passed to wusa.exe. Also occurs in other circumstances. Windows Update Troubleshooter

0x800705B4 / 2147943860 => Timeout Period Expired. Troubleshooting Guide, Windows Update Troubleshooter

0x8024200B / 2149851147 => A hardware update (driver) was not able to be installed. Windows Update Troubleshooter

0x80070020 / 2147942432 = The process cannot access the file because it is being used by another process => Windows update is unable to update specific files due to locking or antivirus. Troubleshooting Guide, Windows Update Troubleshooter

0x80073712 / 2147956498 - The Windows Update component manifest is corrupted. Troubleshooting Guide, Windows Update Troubleshooter

0x80004005 / 2147500037 - Corrupt or missing files in the OS.Reseting Windows Update Components, Windows Update Troubleshooter

0x8024402F / 2149859375 - windows update is having trouble contacting Microsoft servers - could be network settings.

0x80070643 / 2147944003 - problems installing .NET framework update. Troubleshooting Guide Windows Update Troubleshooter

Comprehensive List of Windows Update Errors

Comprehensive Windows Update Error List

Additional Links

Reseting Windows Update Components
Windows Update Troubleshooter

Creating Storage Reports With PowerShell

Joshua King
Follow on Twitter @windosnz

Creating Storage Reports With PowerShell

In your environment, you may have monitoring tools that fire off emails when a drive has reached certain usage thresholds, or you may have other tools calculating these trends.

This is great, but sometimes it's helpful to have a script you can run to get a quick overview of all your servers' hard drives at a particular point in time—whether they're nearly empty, nearly full, or somewhere in-between. Of course, it helps if the output can be fed down the PowerShell pipeline into other scripts.

Too long; won't read

If you hadn't guessed, this post will be discussing a complete script. I'm sure some readers won't need my explanation to understand the how's and why's of it. But if you're wanting to skip to "the goods," you can check out the Gist.

For everyone else, know that I'll be deviating from how I normally do these types of posts. Instead of reading through the finished script line by line, I'll be talking about the logical flow of how it was built.

This may result in parts not fitting together right away, but I promise it will all fall into place by the end.

Herding the servers

The first thing to sort out is getting a collection of all of our servers. Strictly speaking, our script should take this list as an input, allowing us to tweak the list each time the report is run. However, I run this script the same way every time and have elected to "hard code" the generation of this collection.

```
$OrgUnit = 'OU=servers,OU=computers,OU=corp,DC=example,DC=com'
$Servers = Get-ADComputer -Filter {(OperatingSystem -like '*Server*') -and (Enabled -eq $True)} -SearchBase $OrgUnit -Properties ManagedBy
```

That seems like a lot for a simple task, right?

It sure is. I'm being a little over the top with the filtering, but I like doing it all up-front to save some post-processing later.

But what is all of that filtering actually doing?

Firstly, providing an Organizational Unit to the SearchBase parameter means that we're only looking for computer accounts within that location. There's no point in grabbing all of our workstations, laptops, and VDI instances when we don't care about them.

The other filtering means we're only getting accounts that are enabled and whose Operating System name contains the word "Server." This will match things like "Windows Server 2012 R2," meaning I only get our servers and not workstations that have accidentally ended up in the wrong container.

The last thing you'll notice is that we're requesting the ManagedBy property. This is so I know who the caretaker of each server is. Armed with a name, I can tap them on the shoulder—or more likely send them an automated email—if I notice a drive that they should be keeping an eye on is nearly full.

Gathering the disks

Now that we have our servers sorted, we need to do a little digging regarding each local disk attached to them. For this, we'll be querying WMI and then combining the resultant information with details about the parent server.

```
$WmiSplat = @{
    ComputerName = $ComputerName
    Class = 'Win32_LogicalDisk'
    Filter = 'DriveType = 3 AND VolumeName != "RESERVED_PAGING_FILE"'
    Property = 'DeviceID', 'FreeSpace', 'Size', 'VolumeName'
    ErrorAction = 'Stop'
}

$Disks = Get-WmiObject @WmiSplat
```

You'll note that we're using splatting for Get-WmiObject, a technique where you supply parameters to a cmdlet via a hash table. Other than helping to avoid long line length, it's not truly necessary in this case. It is, however, what I prefer to do when using many parameters at once, and it's a good habit to get into.

The parameters themselves are fairly standard. Filtering on DriveType 3 means we're only going to be getting local disks, as opposed to network or removable ones. In my environment, we used to create disks specifically for page files. Not many of them exist anymore, but if they do, I don't want to see them in this report. So, we're filtering them out based on the standard name we gave them.

```
foreach ($Disk in $Disks) {
    $PctUsed = ($Disk.Size - $Disk.FreeSpace) / $Disk.Size

    [PSCustomObject] [Ordered] @{
        ComputerName     = $ComputerName
        ManagedBy        = $ManagedBy
        DriveLetter      = $Disk.DeviceID
        VolumeName       = $Disk.VolumeName
        SizeRemaining    = $Disk.FreeSpace
        SizeRemainingGB = [Math]::Round($Disk.FreeSpace / 1GB, 2)
        Size             = $Disk.Size
        SizeGB           = [Math]::Round($Disk.Size / 1GB, 2)
        Usage            = New-PercentBar -Percent $PctUsed -BarCharacter '▒'
    }
}
```

Now, for each disk, let's wrap up the information we've got into a neat custom object. Most of the properties we're just pulling through directly from the $Disk object. (ComputerName and ManagedBy are coming from the ADComputer object, more on that later.)

Size and SizeGB are two different representations of the same thing; the first measuring the size of a disk in bytes and the second being converted into gigabytes. PowerShell has a nice shorthand for doing this conversion: simply divide the byte value by "1GB," and this works for other units too. We're rounding the resulting figure, as conversions like this can tend to end up with many decimal places.

You may be wondering why [Ordered] is slotted in there between [PSCustomObject] and the hash table defining its properties. By default, hash tables don't have a set order, so even though Usage is defined last, it might end up being displayed first. This is normally fine, but, in this case, I want some control over the output of this script, and [Ordered] is how you tell PowerShell that the order is to be preserved.

Finally, the Usage property is using my PoshPctBar module to display disk usage in a graphical form (for example: [▓▓........]). This is optional and requires installing the module, but I find having this included allows me to quickly pick out problem disks.

What about errors?!

Good catch (pun intended)!

You never know what might cause it—whether it's that you don't have permission on the remote server or WMI isn't responding for whatever reason—but from time to time Get-WmiObject may generate an error instead of "useful" output.

The eagle-eyed reader out there would have noted that the code snippet in the previous section included setting the error action for the WMI cmdlet to "stop." This is to ensure that any errors from that cmdlet are "terminating" and will be caught using try/catch.

To get some workable error handling, wrap the previous snippet in a "try" block, and then we'll be able to mitigate them in a "catch" block.

```
try {
    # Previous snippet
} catch {
    [PSCustomObject] [Ordered] @{
        ComputerName     = $ComputerName
        ManagedBy        = $ManagedBy
        DriveLetter      = $null
        VolumeName       = $null
        SizeRemaining    = $null
        SizeRemainingGB  = $null
        Size             = $null
        SizeGB           = $null
        Usage            = $_.Exception.Message
    }
}
```

With this catch block, we're creating an object with the same properties as what we previously created. This means they'll seamlessly output alongside our "working" objects, but with enough information included so that we know which server the error occurred on, and what the message of the error was. The message is just a string and is being included in the Usage property, which is normally a string anyway.

In the context of a catch block, $_ changes to the error that caused the block to trigger. This is why ComputerName and ManagedBy are being supplied as their own variable, rather than properties of a parent object as you might expect to see them.

My screen isn't that wide!

At this point, we've got a lot of properties for each disk—and a few of them are just two different ways of seeing the same value. By default, when there are this many properties, PowerShell will opt to output this information as a list rather than a nicely formatted table.

Ever notice how you'll often run a cmdlet and only see a subset of the infor mation available unless you pipe the output to Select-Object *?

We're able to do the same thing with our custom objects by specifying a "Default Display Property Set." In short, we'll be telling PowerShell: "Unless I say otherwise, I only want to see this handful of properties."

But first, we need to establish which properties we want to see:

```
$DefaultDisplaySet = 'ComputerName', 'DriveLetter', 'SizeGB', 'Usage'
$DefaultDisplayPropertySet = New-Object System.Management.Automation.PSPropertySet('DefaultDisplayPropertySet', [String[]] $DefaultDisplaySet)
$PSStandardMembers = [System.Management.Automation.PSMemberInfo[]]@($DefaultDisplayPropertySet)
```

Then, we need to "apply" this to each of our objects before pushing them down the pipeline:

```
$DiskObj = [PSCustomObject] [Ordered] @{
    # Previous snippet
}

$DiskObj | Add-Member MemberSet PSStandardMembers $PSStandardMembers
$DiskObj
```

Four is the magic number when PowerShell is deciding between a list or a table. We'll now only see ComputerName, DriveLetter, SizeGB, Usage unless we decide otherwise. The data is still there if we need it.

This is going to take forever to run in my environment!

It probably would if you were to just iterate through each server one at a time. Luckily, there are a few options for running these tasks against more than one target at any given moment.

My go-to for this is PoshRSJob. I use it all the time (perhaps too often). This module simplifies the creation of runspaces, to the point where you barely have to think about (or understand) them.

Most of the previous code, everything except generating a list of servers, gets wrapped up into a script block variable (executable code stored in a variable, similar to a function). As a habit, I refer to this variable as $JobBlock.

```
$JobBlock = {
    if (Test-Connection -ComputerName $_.Name -Count 1 -Quiet) {
        # DefaultDisplayPropertySet

        $ComputerName = $_.Name
        $ManagedBy = ($_.ManagedBy -split ",*..=")[1]

        try {
            # Previous snippet
        } catch {
            # Previous snippet
        }
    }
}
```

One thing you'll note here is the use of $_, which denotes the current server. As mentioned above, some properties that we'll potentially need inside our catch block are being stored as independent variables.

ManagedBy has a little bit of string manipulation going on to make the output more useful for me. In my case, I only want to know the name of the user, rather than their full distinguished name. You may want to adjust this if you require different information.

You'll also notice that we're checking to make sure that each server is online by pinging it once before attempting to gather disk info. This works in my environment because the servers are configured to respond to my workstation.

To get the ball rolling, simply pipe your collection of servers into the PoshRSJob functions as shown:

```
$Servers | Select-Object * | Start-RSJob -ScriptBlock $JobBlock -Name {$_.Name} | Wait-RSJob -ShowProgress | Receive-RSJob
```

The servers are going through Select-Object so that the ManagedBy property is available. Start-RSJob is the function doing the heavy lifting of getting our task done; it's passing each server object into our script block, and the content of the script block is running through for each of them. Just in case, we're naming each job after the name of the given server, allowing us to troubleshoot if anything goes wrong.

Generally, starting a bunch of jobs would just do that—start the jobs and then return control to you so that you can keep working while they complete in the background. That's not what I actually want to happen with this script, so we use the Wait-RSJob function to prevent ourselves from performing other actions until they all complete (or fail), and the ShowProgress switch will give us a visual indicator of how many jobs have completed.

Finally, Receive-RSJob will collect the resultant output from each job.

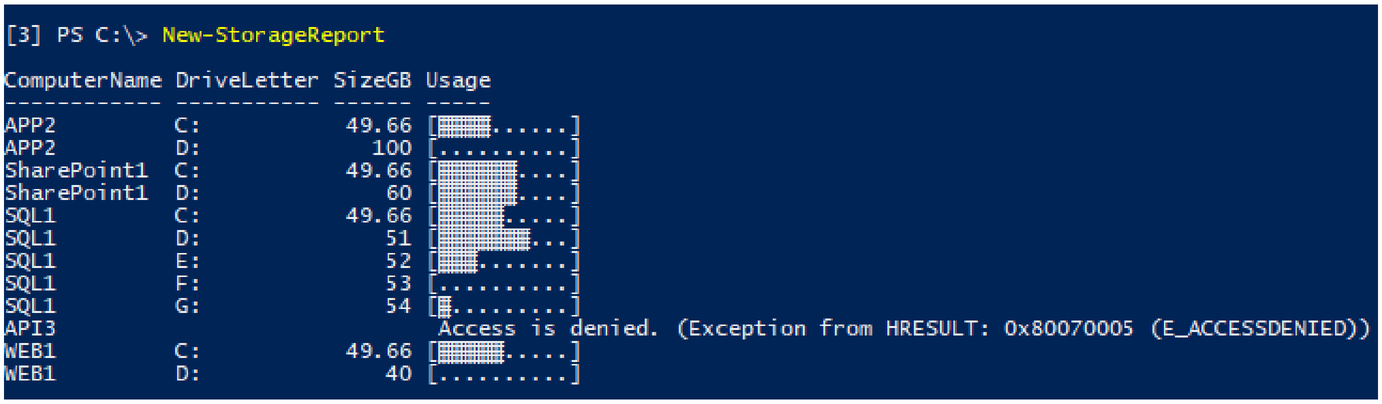
The proof is in the pudding

I've not actually shown you any output from all of this yet; it seemed a little premature until it all comes together at the end.

Make sure you have a look at the final product all stitched together, wrapped up as a function with some comment-based help.

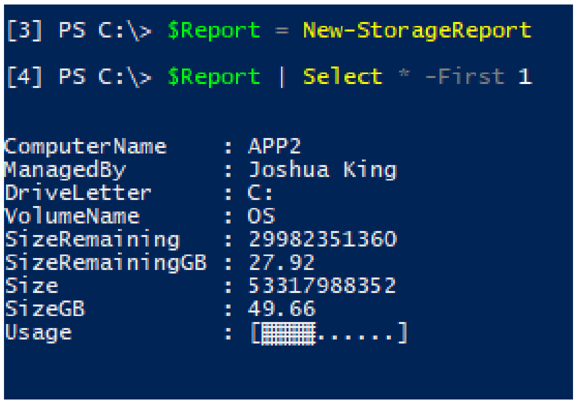
But, how do you actually use the thing?

First, just run it directly and view the output inside your PowerShell host:



```
[3] PS C:\> New-StorageReport

ComputerName DriveLetter SizeGB Usage
------------ ----------- ------ -----
APP2         C:           49.66 [▒▒▒▒......]
APP2         D:             100 [..........]
SharePoint1  C:           49.66 [▒▒▒▒▒▒....]
SharePoint1  D:              60 [▒▒▒▒▒▒....]
SQL1         C:           49.66 [▒▒▒▒▒.....]
SQL1         D:              51 [▒▒▒▒▒▒▒...]
SQL1         E:              52 [▒▒▒.......]
SQL1         F:              53 [..........]
SQL1         G:              54 [▒.........]
API3                             Access is denied. (Exception from HRESULT: 0x80070005 (E_ACCESSDENIED))
WEB1         C:           49.66 [▒▒▒▒▒.....]
WEB1         D:              40 [..........]
```

Next, pop the output into a variable for further manipulation:



```
[3] PS C:\> $Report = New-StorageReport

[4] PS C:\> $Report | Select * -First 1


ComputerName     : APP2
ManagedBy        : Joshua King
DriveLetter      : C:
VolumeName       : OS
SizeRemaining    : 29982351360
SizeRemainingGB  : 27.92
Size             : 53317988352
SizeGB           : 49.66
Usage            : [████......]
```

From here, it’s up to you to tweak as you see fit.

Final thoughts

That was a long journey, looking back at this post, but I hope the result was worth it. Working through this covered a number of topics, and given how many of them there were, I couldn’t dive into them all as deeply as I might have liked to.

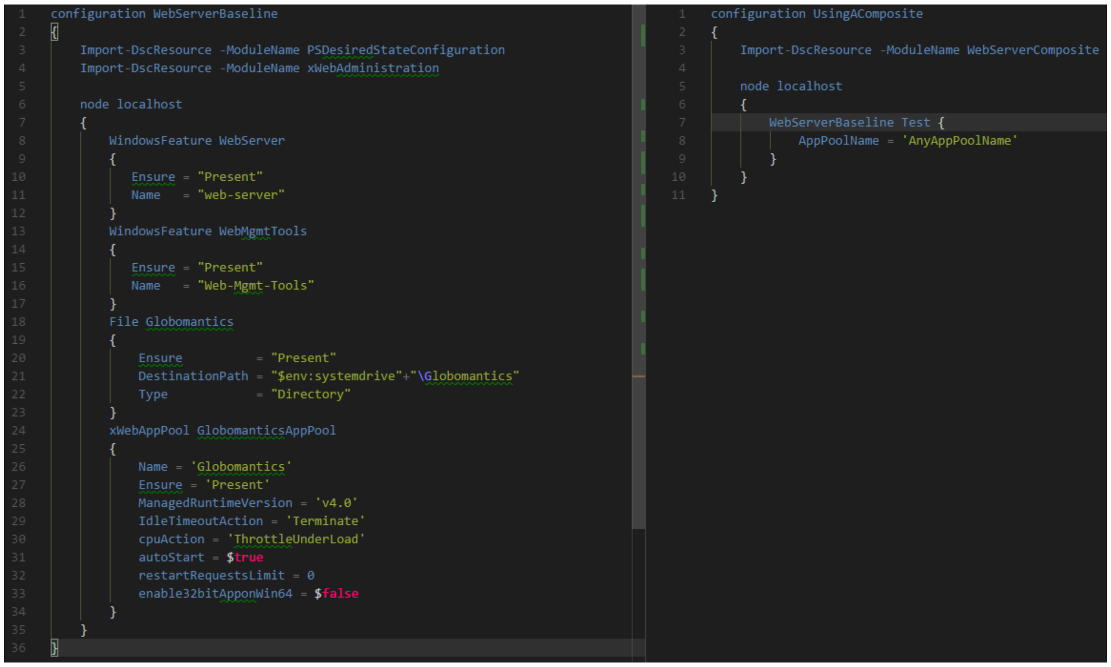
If there’s a specific topic you want covered in more detail in a follow-up post, please let me know here or on Twitter.

And finally, please check out the other #PSBlogWeek posts, and follow the hashtag to find other useful PowerShell content!

Using Desired State Configuration (DSC) Composite Resources

Josh Duffney
Follow on Twitter @joshduffney

Composite resources can be thought of as help functions, but instead of helper functions for your PowerShell scripts it's a helper resource for your DSC configurations. They help solve the same problem helper functions do by modularizing your code. Which reduces the length and complexity of your code. Here is a comparison between a DSC configuration one without using a composite resource and with using a composite resource. As you can probably guess the shorter configuration is the one using a composite resource. Taking some of the logic out of the main DSC configuration helps you maintain the code by breaking it apart.



```
configuration WebServerBaseline
{
    Import-DscResource -ModuleName PSDesiredStateConfiguration
    Import-DscResource -ModuleName xWebAdministration

    node localhost
    {
        WindowsFeature WebServer
        {
           Ensure = "Present"
           Name   = "web-server"
        }
        WindowsFeature WebMgmtTools
        {
           Ensure = "Present"
           Name   = "Web-Mgmt-Tools"
        }
        File Globomantics
        {
            Ensure          = "Present"
            DestinationPath = "$env:systemdrive"+"\Globomantics"
            Type            = "Directory"
        }
        xWebAppPool GlobomanticsAppPool
        {
            Name = 'Globomantics'
            Ensure = 'Present'
            ManagedRuntimeVersion = 'v4.0'
            IdleTimeoutAction = 'Terminate'
            cpuAction = 'ThrottleUnderLoad'
            autoStart = $true
            restartRequestsLimit = 0
            enable32bitApponWin64 = $false
        }
    }
}
```

```
configuration UsingAComposite
{
    Import-DscResource -ModuleName WebServerComposite

    node localhost
    {
        WebServerBaseline Test {
            AppPoolName = 'AnyAppPoolName'
        }
    }
}
```

Understanding Composite Resource

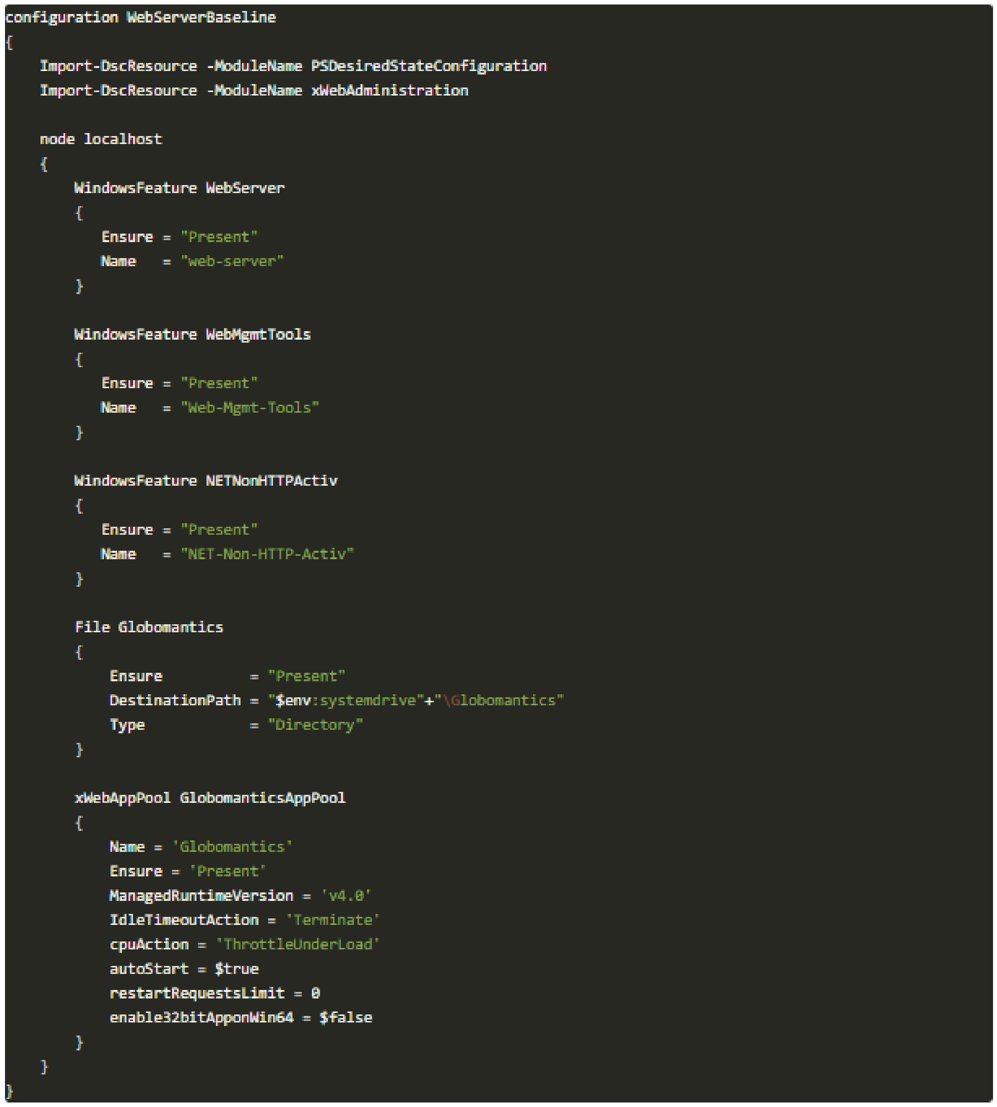
Composite resources work just like mof based resources. They must be placed in the $env:psmodulepath to be used, they even follow a similar folder structure–where you have a resource module folder and then all the resources themselves living under a sub folder called DSCResources. The main difference between a composite and a mof based resource is a composite resource uses a .schema.psm1 file instead of a normal PowerShell module .psm1 file. Composite resource still contain a .psd1 as a manifest, but that simply points to the .schema.psm1.

In the example below, the resource module is called CompositeModule, which is the top-level folder. Under that folder is the CompositeModule.psd1 file, which is the module manifest for the composite module. At the root of the CompositeModule folder is a directory called DSCResources. If you have written DSC resources before, this folder will look familiar. This is the directory that contains all the actual DSC resources and code responsible for making the DSC configuration work. Each composite resource you create will exist under this folder. In the example shown above, there is only one composite resource with the name CompositeResource. Inside that directory are two files: CompositeResource.psd1 and CompositeResource.schema.psm1. CompositeResource.psd1 is the manifest for the composite resource, and the CompositeResource.schema.psm1 file is where all the code goes for your composite resource.

- CompositeModule
    - CompositeModule.psd1
    - DSCResources
        - CompositeResource
            - CompositeResource.psd1
            - CompositeResource.schema.psm1

DSC Without a Composite Resource

Now that you have an understanding of what makes up a composite resource, let's take a look at what a normal DSC configuration looks like. For this example, I have a web server baseline configuration. It uses the built-in DSC module, PSDesiredStateConfiguration, and a custom DSC resource module, xWebAdministration. Since this configuration is a baseline, it needs to be applied to every web server in the environment. However, I have different types of web servers, so this section of code is repeated over and over again in several separate configurations. Just like when writing PowerShell functions or modules, you should condense repeated code if possible, and that's what a composite resource allows us to do.



```
configuration WebServerBaseline
{
    Import-DscResource -ModuleName PSDesiredStateConfiguration
    Import-DscResource -ModuleName xWebAdministration

    node localhost
    {
        WindowsFeature WebServer
        {
           Ensure = "Present"
           Name   = "web-server"
        }

        WindowsFeature WebMgmtTools
        {
           Ensure = "Present"
           Name   = "Web-Mgmt-Tools"
        }

        WindowsFeature NETNonHTTPActiv
        {
           Ensure = "Present"
           Name   = "NET-Non-HTTP-Activ"
        }

        File Globomantics
        {
            Ensure          = "Present"
            DestinationPath = "$env:systemdrive"+"\Globomantics"
            Type            = "Directory"
        }

        xWebAppPool GlobomanticsAppPool
        {
            Name = 'Globomantics'
            Ensure = 'Present'
            ManagedRuntimeVersion = 'v4.0'
            IdleTimeoutAction = 'Terminate'
            cpuAction = 'ThrottleUnderLoad'
            autoStart = $true
            restartRequestsLimit = 0
            enable32bitAppOnWin64 = $false
        }
    }
}
```

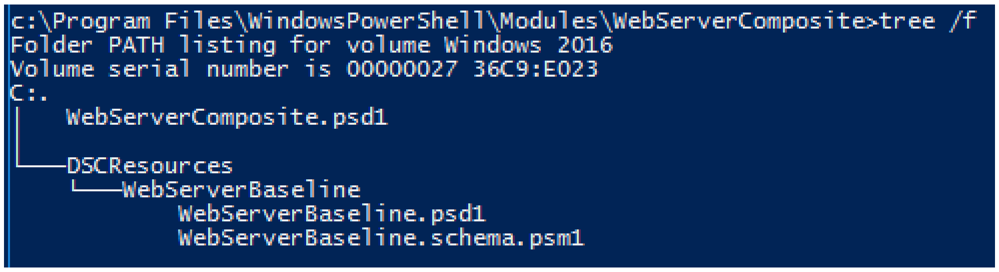
Creating Composite Resources

At this point, we have a baseline configuration we want to condense—that's the problem we're trying to solve with a composite resource. So we have the configuration code, but how do we create a composite resource? Luckily, someone has already solved that problem by creating a helper function that generates the composite resource for you. The helper function can be found on GitHub, and it's called New-DscCompositeResource. After you look at the code, you'll notice several parameters, but the important ones are -Path, -ModuleName, and -ResourceName. All are self explanatory, but make sure you specify a $env:psmodulepath for the path so the resource can be used.

Be sure to load the helper function New-DscCompositeResource in to memory before running the below code

```
$splat = @{
    Path = ($env:PSModulePath -split ';')[1]
    ModuleName = 'WebServerComposite'
    ResourceName = 'WebServerBaseline'
    Author = 'Josh Duffney'
    Company = 'duffneyio'
}

New-DscCompositeResource @splat
```

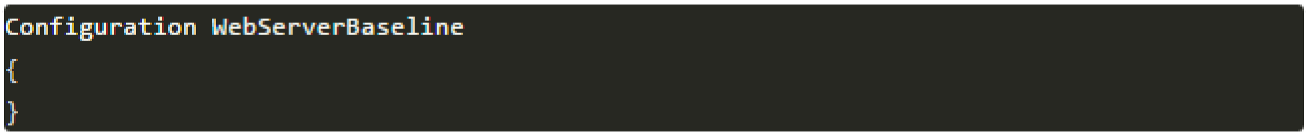
After running the code above, you should see the same folder structure as shown below in the tree output screenshot.



```
c:\Program Files\WindowsPowerShell\Modules\WebServerComposite>tree /f
Folder PATH listing for volume Windows 2016
Volume serial number is 00000027 36C9:E023
C:.
│   WebServerComposite.psd1
│
└───DSCResources
    └───WebServerBaseline
            WebServerBaseline.psd1
            WebServerBaseline.schema.psm1
```

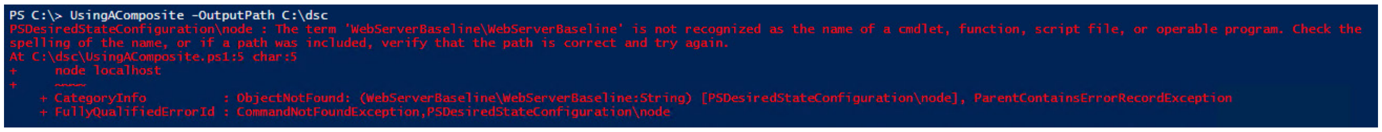
You've now got a composite resource, but it won't do anything because we've not added any logic into the composite resources themselves. The composite resource I created for the web server baseline is called Web-

ServerBaseline. To update that resource, I have to edit the WebServerBaseline.schema.psm1 file. When you open it, the code should look like the configuration shown below.



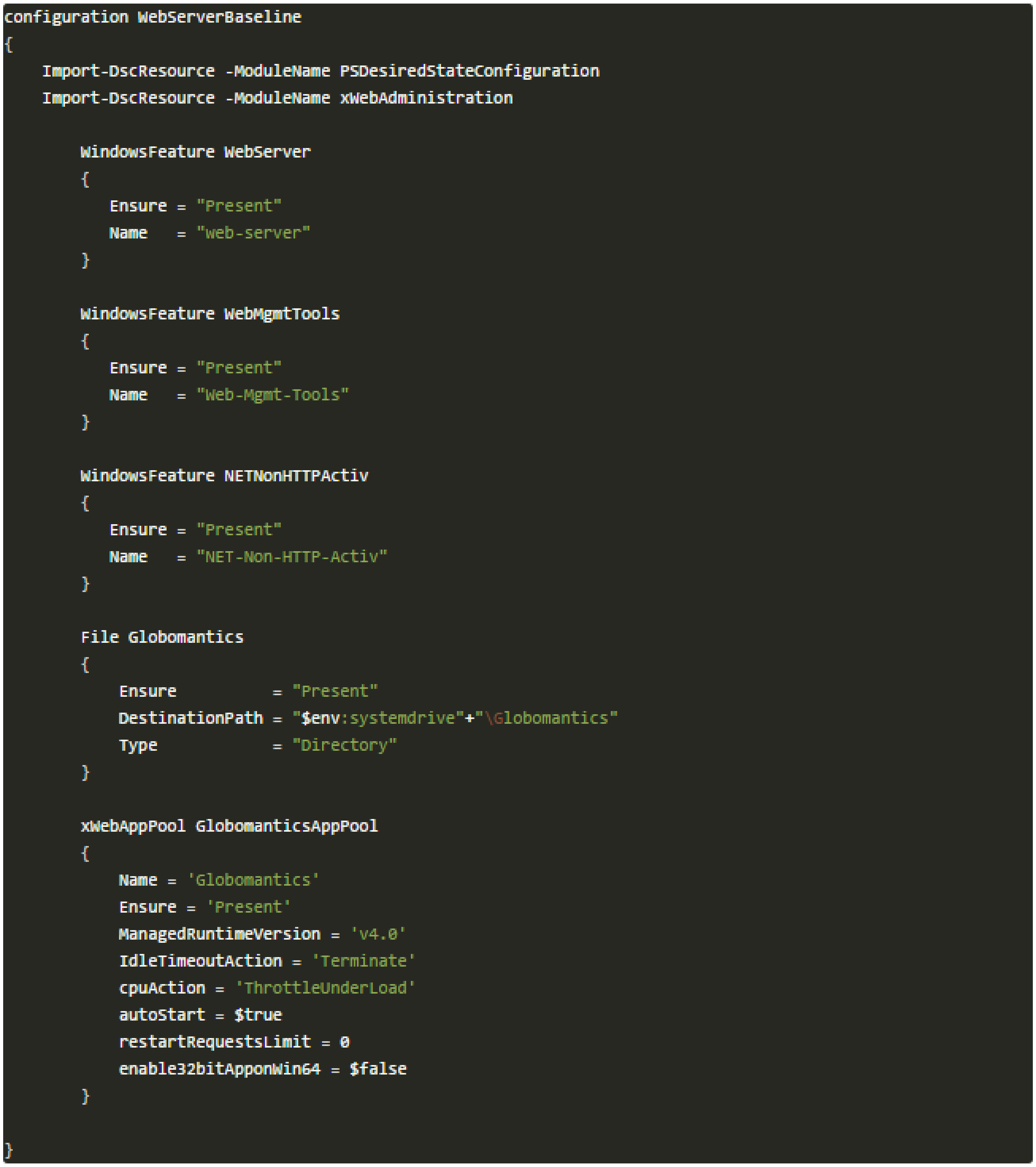
```
Configuration WebServerBaseline
{
}
```

At the moment, it's an empty configuration called WebServerBaseline. Since we already have the DSC code required for this composite, we can simply copy and paste it in. However, there is one change we have to make to the DSC configuration before it can be used in this composite resource. In order for it to work, we have to remove the node block from the configuration. If we don't, we'll get an error when trying to compile the MOF.



```
PS C:\> UsingAComposite -OutputPath C:\dsc
PSDesiredStateConfiguration\node : The term 'WebServerBaseline\WebServerBaseline' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the
spelling of the name, or if a path was included, verify that the path is correct and try again.
At C:\dsc\UsingAComposite.ps1:5 char:5
+     node localhost
+     ~~~~
    + CategoryInfo          : ObjectNotFound: (WebServerBaseline\WebServerBaseline:String) [PSDesiredStateConfiguration\node], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : CommandNotFoundException,PSDesiredStateConfiguration\node
```

After removing the node block, the WebServerBaseline.schema.psm1 file should look exactly like the code snippet below.



```
configuration WebServerBaseline
{
    Import-DscResource -ModuleName PSDesiredStateConfiguration
    Import-DscResource -ModuleName xWebAdministration

        WindowsFeature WebServer
        {
           Ensure = "Present"
           Name   = "web-server"
        }

        WindowsFeature WebMgmtTools
        {
           Ensure = "Present"
           Name   = "Web-Mgmt-Tools"
        }

        WindowsFeature NETNonHTTPActiv
        {
           Ensure = "Present"
           Name   = "NET-Non-HTTP-Activ"
        }

        File Globomantics
        {
            Ensure          = "Present"
            DestinationPath = "$env:systemdrive"+"\Globomantics"
            Type            = "Directory"
        }

        xWebAppPool GlobomanticsAppPool
        {
            Name = 'Globomantics'
            Ensure = 'Present'
            ManagedRuntimeVersion = 'v4.0'
            IdleTimeoutAction = 'Terminate'
            cpuAction = 'ThrottleUnderLoad'
            autoStart = $true
            restartRequestsLimit = 0
            enable32bitApponWin64 = $false
        }

}
```

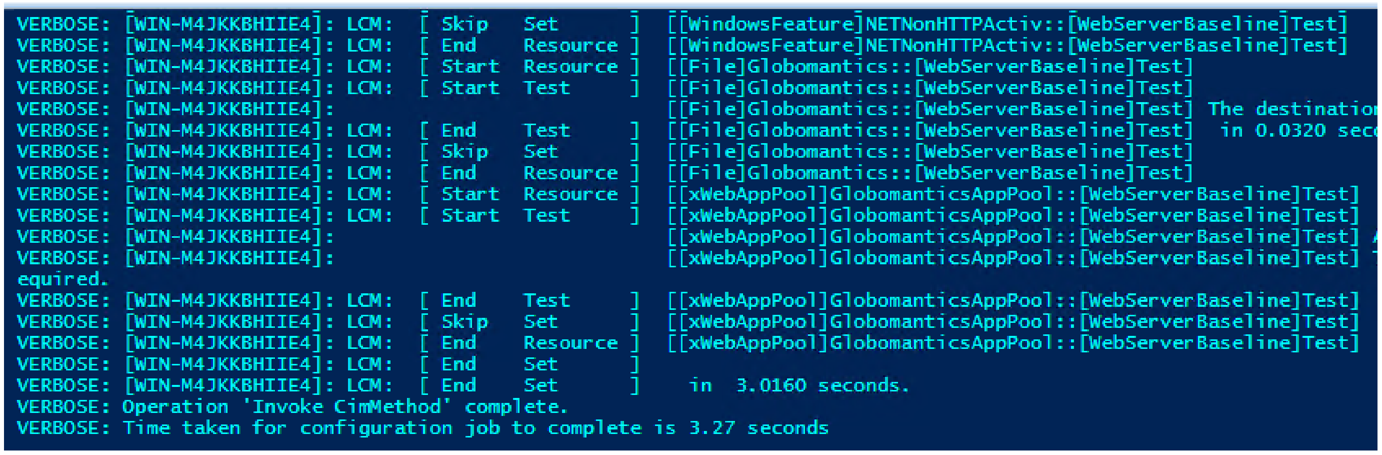
Using Composite Resources

Now that we've updated the WebServerBaseline composite resource, it's now time to write a new DSC configuration that uses that composite resource. For this example, I'll name the new configuration UsingAComposite. The first thing you should do is import the composite module. You do that the same way as you would a normal DSC resource module with the Import-DscResource cmdlet. Right after the Import-DscResource cmdlet, create a node block. I'll just specify local host for my node. Inside the node block is where you declare the composite resource you want to use. In our case, it is WebServerBaseline. Now the next part might throw you off a bit. Inside the WebServerBaseline resource, I do not define any properties. I'll get to why in a minute, but if you take a look at the syntax for the WebServerBaseline resource Get-DscResource webserverbaseline -Syntax, you'll notice there are only two options: DependsOn and PsDscRunAsCredential. Both of those are optional, and I don't need to specify them in my configuration; it will work without them. Notice that the configuration is only 10 lines now, not 45?

```
configuration UsingAComposite
{
    Import-DscResource -ModuleName WebServerComposite

    node localhost
    {
        WebServerBaseline Test {
        }
    }
}
```

The next thing to do is to apply the configuration to make sure it executes properly. To do that, you must load the configuration into memory, generate the MOF document, and then apply the configuration. For this example, I'll just use Push mode and issue a Start-DscConfiguration command. I have the UsingAComposite.ps1 file saved on my target machine, so I can just dot source it in if I have the configuration open in vscode, or I can just load it into memory there. Once the configuration is loaded, I can generate the MOF file by calling the configuration. I specified the outputpath parameter because I wanted to specify the directory the MOF would end up in. After that, apply the configuration to your target server. As I mentioned, I'll do this with the Start-DscConfiguration cmdlet.

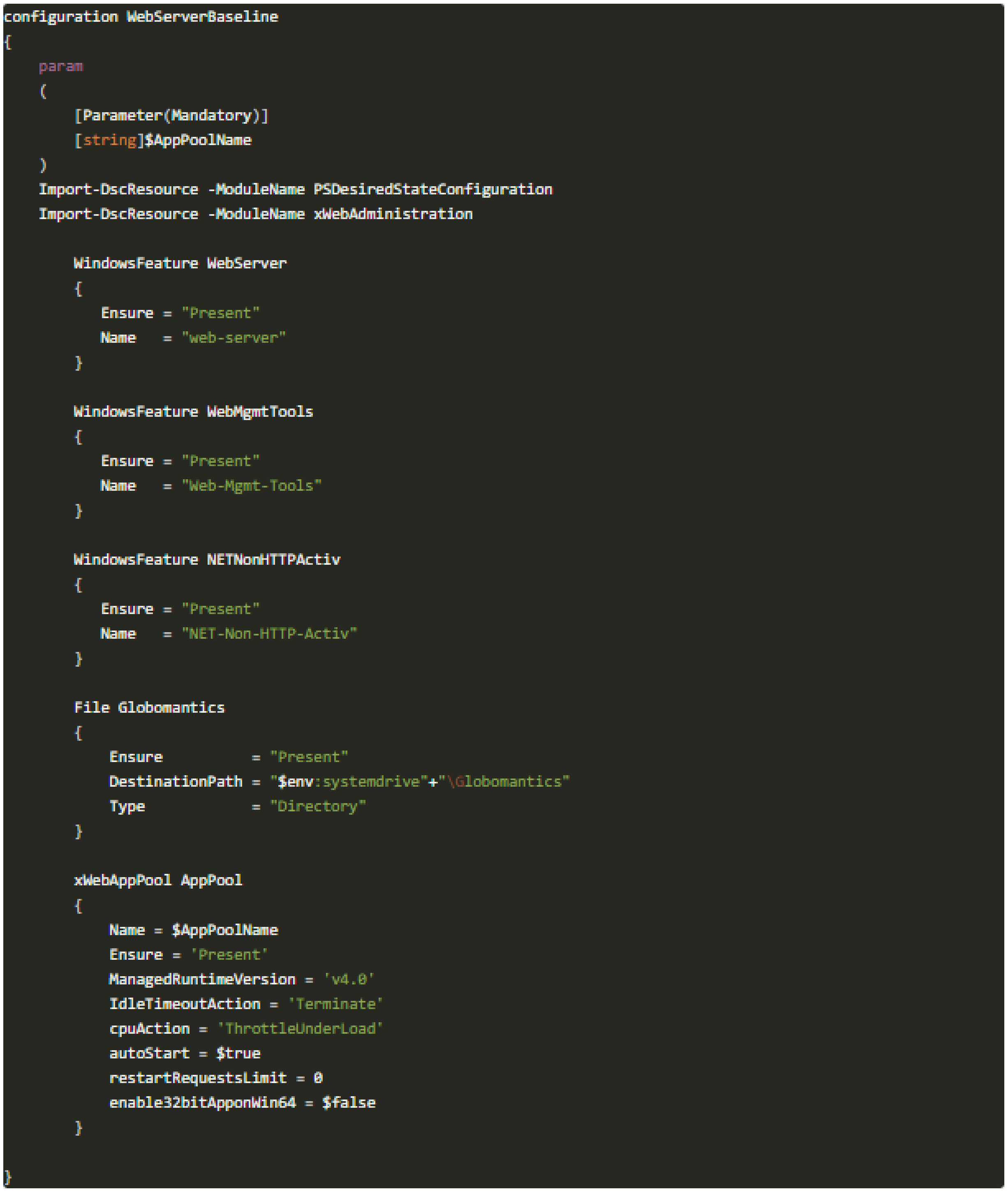
```
. .\UsingAComposite.ps1
UsingAComposite -OutputPath C:\DSC
Start-DscConfiguration -Path C:\DSC -Wait -Verbose
```



```
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Skip   Set      ]  [[WindowsFeature]NETNonHTTPActiv::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Resource ]  [[WindowsFeature]NETNonHTTPActiv::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Start  Resource ]  [[File]Globomantics::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Start  Test     ]  [[File]Globomantics::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]:                            [[File]Globomantics::[WebServerBaseline]Test] The destinatio
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Test     ]  [[File]Globomantics::[WebServerBaseline]Test]  in 0.0320 sec
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Skip   Set      ]  [[File]Globomantics::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Resource ]  [[File]Globomantics::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Start  Resource ]  [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Start  Test     ]  [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]:                            [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]:                            [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
equired.
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Test     ]  [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ Skip   Set      ]  [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Resource ]  [[xWebAppPool]GlobomanticsAppPool::[WebServerBaseline]Test]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Set      ]
VERBOSE: [WIN-M4JKKBHIIE4]: LCM:  [ End    Set      ]    in  3.0160 seconds.
VERBOSE: Operation 'Invoke CimMethod' complete.
VERBOSE: Time taken for configuration job to complete is 3.27 seconds
```

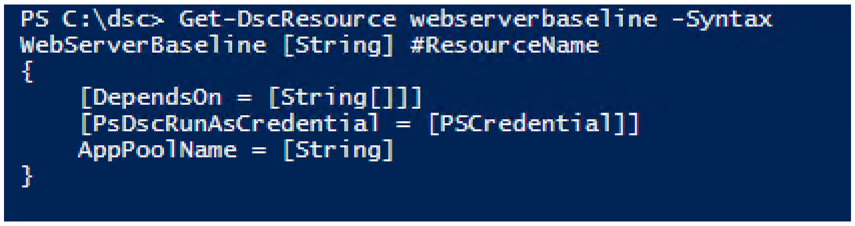
Adding Properties to a Composite Resource

Typically, DSC resources have at least one mandatory property, but the WebServerBaseline composite resource we wrote doesn't. The reason for that is I didn't include any mandatory parameters in the composite configuration when I updated the WebServerBaseline.schema.psm1 file. Parameters are how you create both optional and mandatory properties for your composite resources. In order for us to create a mandatory property, we'll have to update the WebServerBaseline.schema.psm1 file with a parameter. As an example, let's say Globomantics isn't the only possible app pool name for a web server baseline. Therefore, we want to make that a value we can input as a parameter, but we also want to make it mandatory because the configuration will fail without it. In order to do that, we simply add a mandatory parameter to the WebServerBaseline.schema.psm1 file called $AppPoolName.



```
configuration WebServerBaseline
{
    param
    (
        [Parameter(Mandatory)]
        [string]$AppPoolName
    )
    Import-DscResource -ModuleName PSDesiredStateConfiguration
    Import-DscResource -ModuleName xWebAdministration

        WindowsFeature WebServer
        {
           Ensure = "Present"
           Name   = "web-server"
        }

        WindowsFeature WebMgmtTools
        {
           Ensure = "Present"
           Name   = "Web-Mgmt-Tools"
        }

        WindowsFeature NETNonHTTPActiv
        {
           Ensure = "Present"
           Name   = "NET-Non-HTTP-Activ"
        }

        File Globomantics
        {
            Ensure          = "Present"
            DestinationPath = "$env:systemdrive"+"\Globomantics"
            Type            = "Directory"
        }

        xWebAppPool AppPool
        {
            Name = $AppPoolName
            Ensure = 'Present'
            ManagedRuntimeVersion = 'v4.0'
            IdleTimeoutAction = 'Terminate'
            cpuAction = 'ThrottleUnderLoad'
            autoStart = $true
            restartRequestsLimit = 0
            enable32bitApponWin64 = $false
        }

}
```

Now, when I run Get-DscResource webserverbaseline -Syntax, I see a new property called AppPoolName



```
PS C:\dsc> Get-DscResource webserverbaseline -Syntax
WebServerBaseline [String] #ResourceName
{
    [DependsOn = [String[]]]
    [PsDscRunAsCredential = [PSCredential]]
    AppPoolName = [String]
}
```

Because we updated the composite resource, we of course have to update our DSC configuration that uses that composite resource:

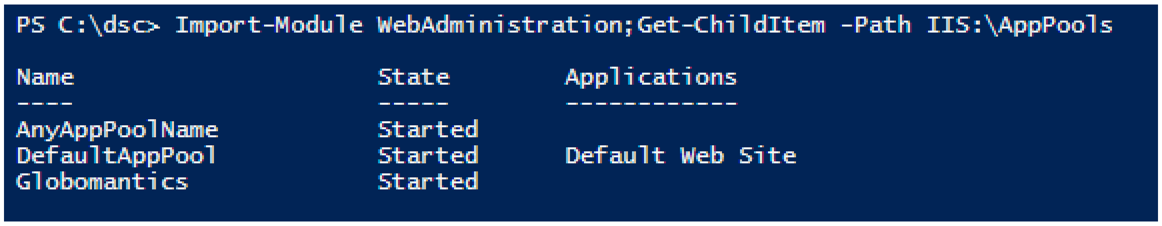
```
configuration UsingAComposite
{
    Import-DscResource -ModuleName WebServerComposite

    node localhost
    {
        WebServerBaseline Test {
            AppPoolName = 'AnyAppPoolName'
        }
    }
}
```

And lastly, if we want to apply this new configuration, we'll have to re-load the configuration into memory, generate a MOF document, and then apply the configuration. The only change should be to add a new app pool with the name of AnyAppPoolName.

```
. .\UsingAComposite.ps1
UsingAComposite -OutputPath c:\DSC
Start-DscConfiguration -Path C:\DSC -Wait -Verbose -Force
```

To check if the app pool got created, import the WebAdministration module and get the contents of IIS:\AppPools:

```
Import-Module WebAdministration;Get-ChildItem -Path IIS:\AppPools
```



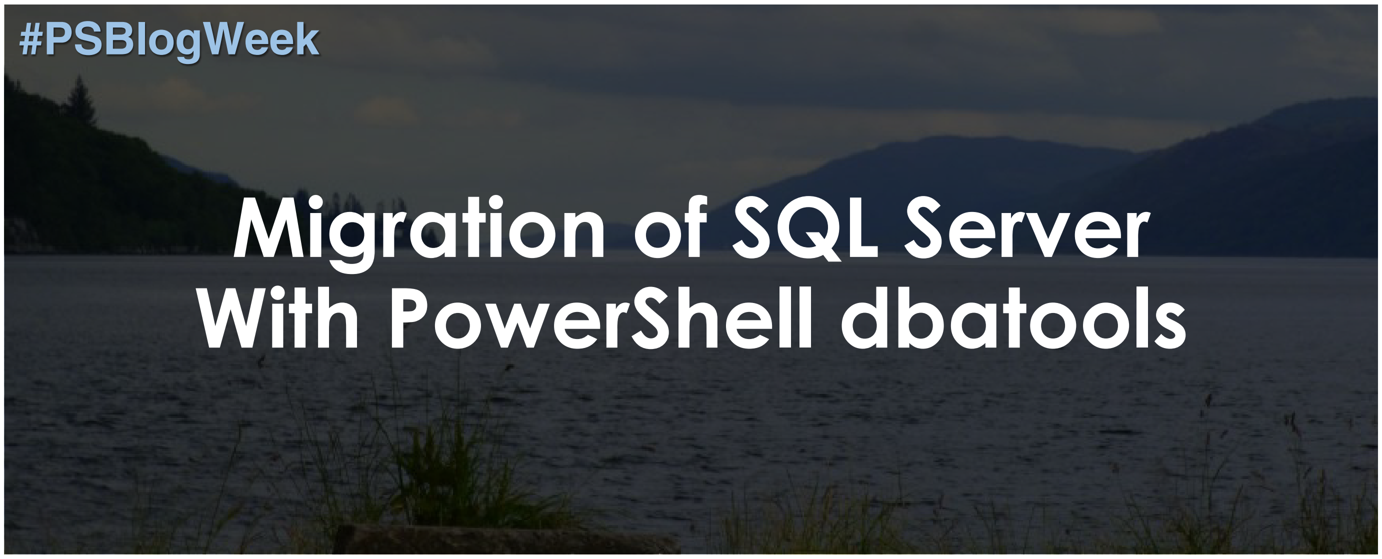
```
PS C:\dsc> Import-Module WebAdministration;Get-ChildItem -Path IIS:\AppPools

Name                     State        Applications
----                     -----        ------------
AnyAppPoolName           Started
DefaultAppPool           Started      Default Web Site
Globomantics             Started
```

Summary

In the end it is up to you to decide if you want to use DSC composite resources or not. They have their advantages as you've seen in this post, but they also have a disadvantage. Which is another set of code in this case a composite PowerShell module that needs deployed to all your target nodes.

Migration of SQL Server With PowerShell dbatools

Volker Bachmann
Follow on Twitter @VolkerBachmann



Migration of SQL Server With PowerShell dbatools

This article is about **server management** with PowerShell and is part of the #PSBlogWeek series (http://psblogweek.com) , created by Adam Bertram.

Index:

It is also part of my blog series about m**igrating our physical SQL Server to a VMware Environment**. For now, all of these articles are in German only – sorry. The first three articles describe the basic server configuration, installation, and VM guest configuration of the VMware Environment. This article describes the migration itself.
I'll write a recap of the whole series in English later on.

1. Introduction to dbatools

I got in contact with PowerShell some years ago, but I wasn't satisfied with what needed to be done to maintain SQL Server.

However, Microsoft has made a lot of improvements since then, and with contributions from several PowerShell Experts and MVPs – such as Chrissy LeMaire, Claudio Silva, Rob Sewell, Constantine Kokkinos and many more, there is now a module that helps to maintain SQL Server 2005+. It's called dbatools, and you can find it here https://dbatools.io. The project is hosted on GitHub and the module is available totally free of charge!

The dbatools community has grown to over 50 contributors with more than 300 SQL Server best practice, administration and migration commands. An overview of the commands can be found here: https://dbatools.io/functions/.

2. Migration Prerequisites

Now, let's turn our attention to the prerequisites for the migration of a physical SQL Server 2008 to a VMware-based SQL Server 2016 on Windows Server 2016. The positive thing here was that there was no need to reinstall everything on the same physical hardware over the weekend. Instead, we bought a totally new VMware Environment with three Dell servers, two network switches, and new storage. There was enough time to test the new SQL Server, the SAN, and build a good configuration for the virtual machines. Most of the VM configuration is based on the blog series „Stairway to Server Virtualization" by David Klee, which can be found on SQL Server Central.

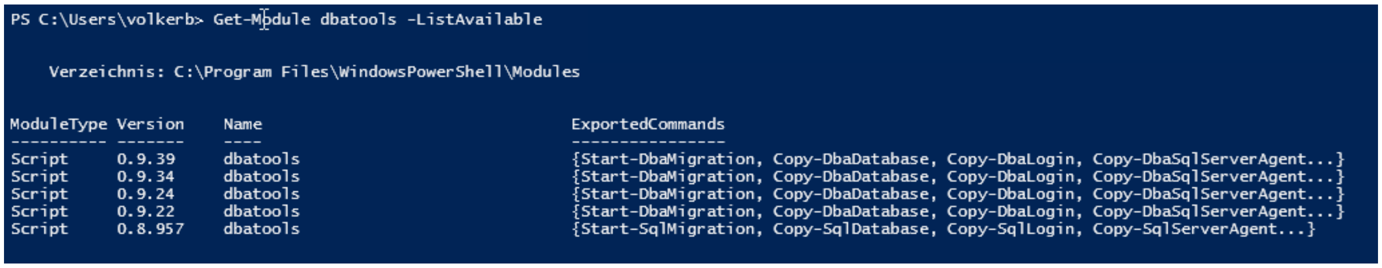
For migration purposes, we installed an additional Windows Server 2016 with PowerShell 5, with SQL Server 2016 as an admin workstation. On the SQL Server, we installed the dbatools by using the easy install-module command:

```
# allow execution of scripts that are signed from a trusted author, if not already set
Set-ExecutionPolicy RemoteSigned

# install
install-module dbatools
```

During installation, you may get a confirmation dialog prompting you to accept installation of the NuGet Package Manager. You should accept; otherwise, you'll need another installation option. These options are described on the dbatools website: https://dbatools.io/download.

The dbatools module is in permanent development – meanwhile, they are near the first major release 1.0 – so you should check for the latest version and update often. Updating is as easy as the installation:

```
Update-dbatools
# Get version number
Get-Module -Name dbatools
# Get all available versions that are installed
Get-Module dbatools -ListAvailable
```



```
PS C:\Users\volkerb> Get-Module dbatools -ListAvailable


    Verzeichnis: C:\Program Files\WindowsPowerShell\Modules


ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.9.39     dbatools                            {Start-DbaMigration, Copy-DbaDatabase, Copy-DbaLogin, Copy-DbaSqlServerAgent...}
Script     0.9.34     dbatools                            {Start-DbaMigration, Copy-DbaDatabase, Copy-DbaLogin, Copy-DbaSqlServerAgent...}
Script     0.9.24     dbatools                            {Start-DbaMigration, Copy-DbaDatabase, Copy-DbaLogin, Copy-DbaSqlServerAgent...}
Script     0.9.22     dbatools                            {Start-DbaMigration, Copy-DbaDatabase, Copy-DbaLogin, Copy-DbaSqlServerAgent...}
Script     0.8.957    dbatools                            {Start-SqlMigration, Copy-SqlDatabase, Copy-SqlLogin, Copy-SqlServerAgent...}
```

On the screenshot we see five versions of the tools installed, so we have to activate the latest version with the comand *import-module*.

```
#Import specific version - here 0.9.39
import-module
"c:\Program Files\WindowsPowerShell\Modules\dbatools\0.9.39\dbatools.psd
1"

# List all available commands of the loaded module version
Get-Command -module dbatools
```

With the last command above you get a quick overview of all the dbatools commands.

After installation of the base SQL Server VM we need to check some basic configuration options first. dbatools can help us with this as well.

All commands are created by experts with references to the corresponding articles where the code comes from.

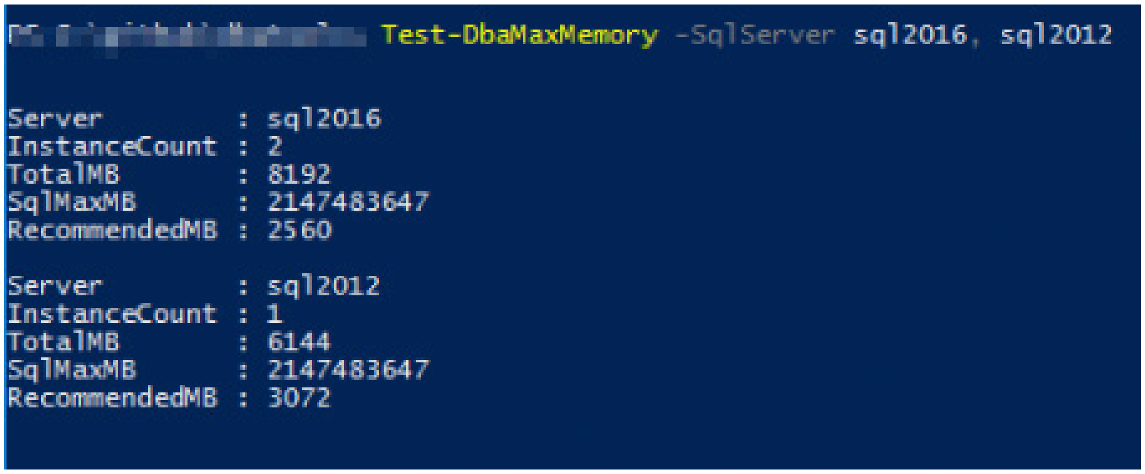
3. Best Practices

- **Max Memory**
  - Test-DbaMaxMemory

- This tests the actual max memory setting against the recommended setting.
-

```
Test-DbaMaxMemory -SqlServer sqlserver01
# command to set the max memory to the recommended
Set-DbaMaxMemory -SqlServer sqlserver01
# or to a fixed value of 2 GB
Set-DbaMaxMemory -SqlInstance sqlserver01 -MaxMb 2048
```

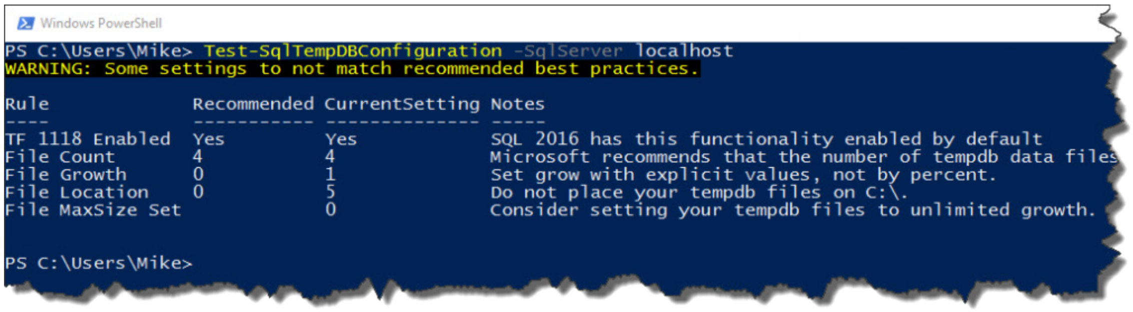
-



```
[redacted] Test-DbaMaxMemory -SqlServer sql2016, sql2012


Server         : sql2016
InstanceCount  : 2
TotalMB        : 8192
SqlMaxMB       : 2147483647
RecommendedMB  : 2560

Server         : sql2012
InstanceCount  : 1
TotalMB        : 6144
SqlMaxMB       : 2147483647
RecommendedMB  : 3072
```

TempDB

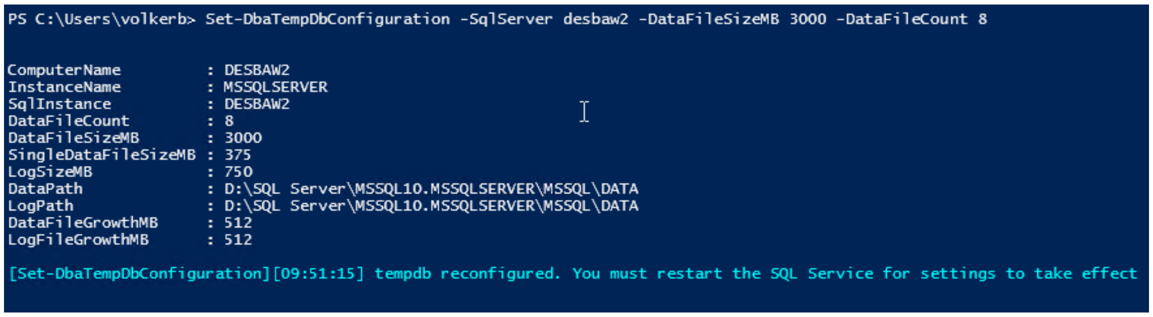
- Test-DbaTempDbConfiguration
- With SQL Server 2016, you get the option to configure the tempdb configuration during installation, but not with older ver sions. With this command, you can control and later adjust it.
- Evaluates tempdb against a set of rules to match best practices. The rules are:
  – TF 1118 Enabled: Is Trace Flag 1118 enabled? (See KB328551)
  – File Count: Does the count of data files in tempdb match the number of logical cores, up to 8?
  – File Growth: Are any files set to have percentage growth? Best practice is that all files have an explicit growth value.
  – **File Location**: Is tempdb located on the C:\ drive? Best prac tice says to locate it elsewhere.
  – **File MaxSize Set (optional)**: Do any files have a max size value? Max size could cause tempdb problems if it isn't allowed to grow.

- 

```
PS C:\Users\Mike> Test-SqlTempDBConfiguration -SqlServer localhost
WARNING: Some settings to not match recommended best practices.

Rule              Recommended CurrentSetting Notes
----              ----------- -------------- -----
TF 1118 Enabled   Yes         Yes            SQL 2016 has this functionality enabled by default
File Count        4           4              Microsoft recommends that the number of tempdb data files
File Growth       0           1              Set grow with explicit values, not by percent.
File Location     0           5              Do not place your tempdb files on C:\.
File MaxSize Set              0              Consider setting your tempdb files to unlimited growth.

PS C:\Users\Mike>
```

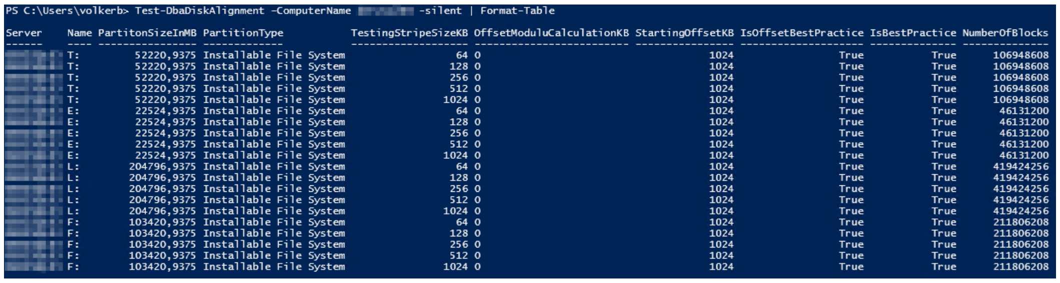
The right configuration can be set by using the corresponding Set- command

A service restart is necessary after reconfiguration, see following screenshot:

- 

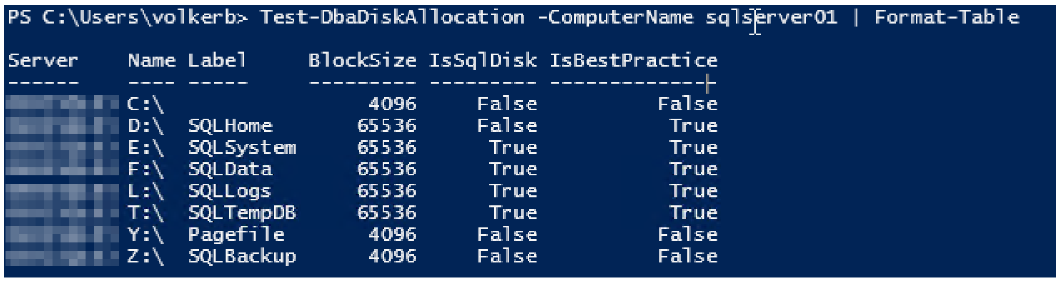
```
PS C:\Users\volkerb> Set-DbaTempDbConfiguration -SqlServer desbaw2 -DataFileSizeMB 3000 -DataFileCount 8


ComputerName         : DESBAW2
InstanceName         : MSSQLSERVER
SqlInstance          : DESBAW2
DataFileCount        : 8
DataFileSizeMB       : 3000
SingleDataFileSizeMB : 375
LogSizeMB            : 750
DataPath             : D:\SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA
LogPath              : D:\SQL Server\MSSQL10.MSSQLSERVER\MSSQL\DATA
DataFileGrowthMB     : 512
LogFileGrowthMB      : 512

[Set-DbaTempDbConfiguration][09:51:15] tempdb reconfigured. You must restart the SQL Service for settings to take effect
```

Disk

- **Test-DbaDiskAlignment**
  - This command verifies that your non-dynamic disks are aligned according to physical requirements.
  - Test-DbaDiskAlignment -ComputerName sqlserver01 | Format-Table
  - 

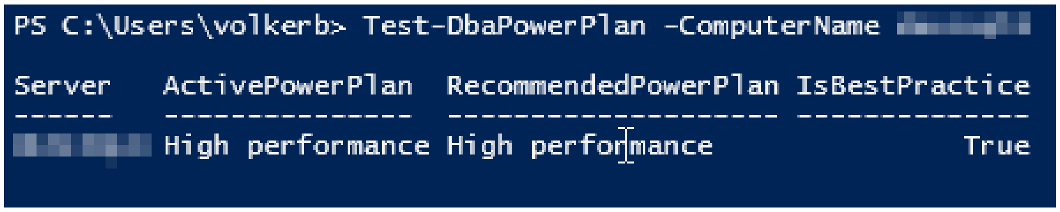
```
PS C:\Users\volkerb> Test-DbaDiskAlignment -ComputerName [redacted] -silent | Format-Table

Server     Name PartitonSizeInMB PartitionType            TestingStripeSizeKB OffsetModuluCalculationKB StartingOffsetKB IsOffsetBestPractice IsBestPractice NumberOfBlocks
------     ---- ---------------- -------------            ------------------- ------------------------- ---------------- -------------------- -------------- --------------
[redacted] T:         52220,9375 Installable File System                   64 0                                     1024                 True           True      106948608
[redacted] T:         52220,9375 Installable File System                  128 0                                     1024                 True           True      106948608
[redacted] T:         52220,9375 Installable File System                  256 0                                     1024                 True           True      106948608
[redacted] T:         52220,9375 Installable File System                  512 0                                     1024                 True           True      106948608
[redacted] T:         52220,9375 Installable File System                 1024 0                                     1024                 True           True      106948608
[redacted] E:         22524,9375 Installable File System                   64 0                                     1024                 True           True       46131200
[redacted] E:         22524,9375 Installable File System                  128 0                                     1024                 True           True       46131200
[redacted] E:         22524,9375 Installable File System                  256 0                                     1024                 True           True       46131200
[redacted] E:         22524,9375 Installable File System                  512 0                                     1024                 True           True       46131200
[redacted] E:         22524,9375 Installable File System                 1024 0                                     1024                 True           True       46131200
[redacted] L:        204796,9375 Installable File System                   64 0                                     1024                 True           True      419424256
[redacted] L:        204796,9375 Installable File System                  128 0                                     1024                 True           True      419424256
[redacted] L:        204796,9375 Installable File System                  256 0                                     1024                 True           True      419424256
[redacted] L:        204796,9375 Installable File System                  512 0                                     1024                 True           True      419424256
[redacted] L:        204796,9375 Installable File System                 1024 0                                     1024                 True           True      419424256
[redacted] F:        103420,9375 Installable File System                   64 0                                     1024                 True           True      211806208
[redacted] F:        103420,9375 Installable File System                  128 0                                     1024                 True           True      211806208
[redacted] F:        103420,9375 Installable File System                  256 0                                     1024                 True           True      211806208
[redacted] F:        103420,9375 Installable File System                  512 0                                     1024                 True           True      211806208
[redacted] F:        103420,9375 Installable File System                 1024 0                                     1024                 True           True      211806208
```

- **Test-DbaDiskAllocation**

- Checks all disks on a computer to see if they are formatted to 64k block size, the best practice for SQL Server disks.
- Test-DbaDiskAllocation -ComputerName sqlserver01 | Format-Table
- 

```
PS C:\Users\volkerb> Test-DbaDiskAllocation -ComputerName sqlserver01 | Format-Table

Server     Name Label      BlockSize IsSqlDisk IsBestPractice
------     ---- -----      --------- --------- --------------
[redacted] C:\                  4096     False          False
[redacted] D:\  SQLHome        65536     False           True
[redacted] E:\  SQLSystem      65536      True           True
[redacted] F:\  SQLData        65536      True           True
[redacted] L:\  SQLLogs        65536      True           True
[redacted] T:\  SQLTempDB      65536      True           True
[redacted] Y:\  Pagefile        4096     False          False
[redacted] Z:\  SQLBackup       4096     False          False
```

PowerPlan

- **Test-DbaPowerPlan**
  - TThe Power Plan should be set to High Performance on every SQL Server
  - Test-DbaPowerPlan -ComputerName sqlserver01
  - 

```
PS C:\Users\volkerb> Test-DbaPowerPlan -ComputerName [redacted]

Server     ActivePowerPlan  RecommendedPowerPlan IsBestPractice
------     ---------------  -------------------- --------------
[redacted] High performance High performance               True
```

SPN

• We use DNS CNAMEs for referring to our SQL Server (See the article „Using Friendly Names for SQL Servers via DNS“ below). We need to adjust the SPN settings manually. That is easy with these commands: Get-DbaSpn and Set-DbaSPN

SQL Server Name

• We created a Single VM template where all SQL Server are created from. With CPU, Memory and Disk Layout as described in the Stairway I mentioned above (1).

• After creating a new VM out of the template the server name changes but the internal SQL Server name does not. Help comes again with dbatools command Repair-DbaServerName
Works fine for me!

4. Migration

• Now for the best part – the migration itself. You normally only need a single command to migrate everything from one SQL Server to another. As described in the Help documentation, this is a „one-button click".
Start-DbaMigration -Source sql2014 -Destination sql2016 -BackupRestore -NetworkShare \nas\sql\migration

• This migrates the follwing parts as listed below. Every part can be skipped with a -no*** parameter as described in the Help documentation – for example, use -NoLogins if you don't want to transfer the logins.

- SQL Server configuration
- Custom errors (user-defined messages)
- SQL credentials
- Database mail
- User objects in system databases
- Central Management Server
- Backup devices
- Linked server
- System triggers
- Databases
- Logins
- Data collector collection sets
- Audits
- Server audit specifications
- Endpoints
- Policy management
- Resource Governor
- Extended Events
- JobServer = SQL Server Agent

• If any error comes up, use the functions, that are called out of the Start-DbaMigration commands step by step.

• Keep in mind that the server configuration is also part of the migration, so min and max memory and all other parameter in sp_configure are transferred. If you want to keep this settings as set by the best practices commands, you should skip the configuration during transfer. Use -NoSpConfigure!

• So what is missing in the moment?
  • Most of the special parts of the additional services:
    • SSIS
    • SSAS
    • SSRS

• You can test the whole migration with the -WhatIf parameter, which shows what's working and what isn't. Sometimes the connection to the target computer isn't working because PowerShell remoting is not enabled (see above).

There is a command to test the connection to the server, and you can find that here:

https://dbatools.io/functions/test-dbacmconnection
There is no need for updating the new server to the latest version of PowerShell, Version 3.0 is enough.

• The whole command looks like this for me:
  • *Start-SqlMigration -Verbose -Source desbsql1 -Destination desbaw2 -BackupRestore -NetworkShare \\DESBAW2\Transfer -DisableJob sOnDestination -Force*

• The parameter DisableJobsOnDestination is extremly helpful when you go to the next step and test the migration itself. When you do this more than once, you also need the parameter –Force, which overwrites the target objects (logins, databases and so on) if they exist from a previous test.

• The parameter -Verbose is useful when an error comes up and you need to dig deeper into the problem.

• Before we wrap up, her's a link to a YouTube video that shows how fast the migration works. Of course it's all going to depend on the size of your databases:

https://youtu.be/PciYdDEBiDM

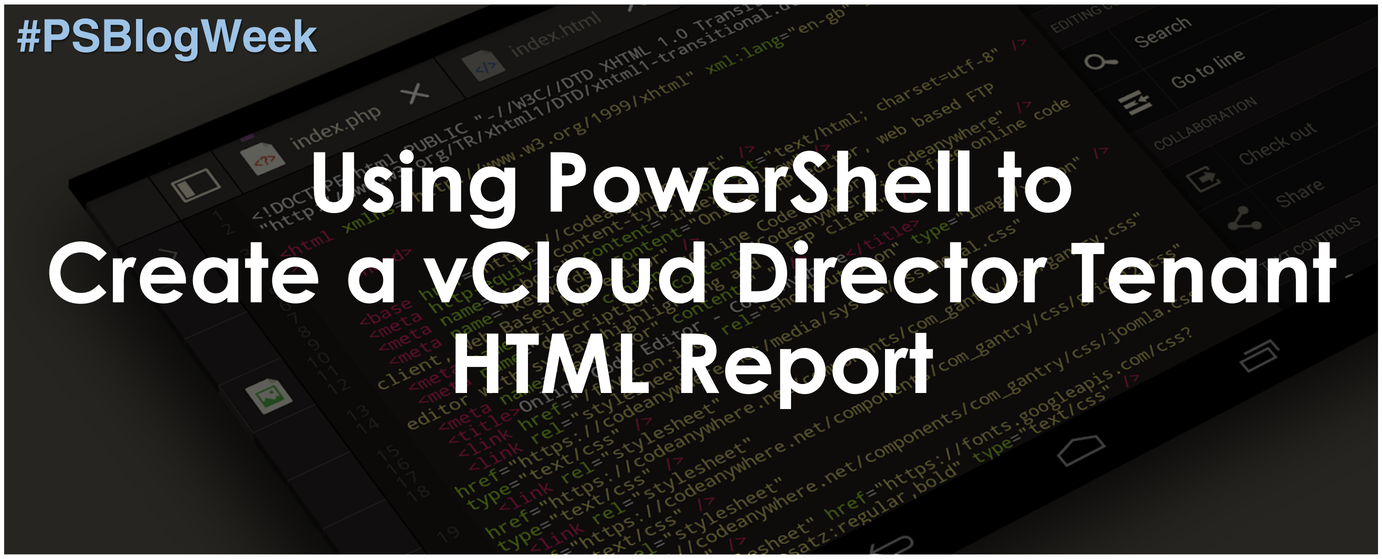
5. References:

1. Stairway to SQL Server Virtualization by David Klee
2. Using Friendly Names for SQL Servers via DNS

Thanks for reading,
Volker Bachmann

Using PowerShell to Create a vCloud Director Tenant HTML Report

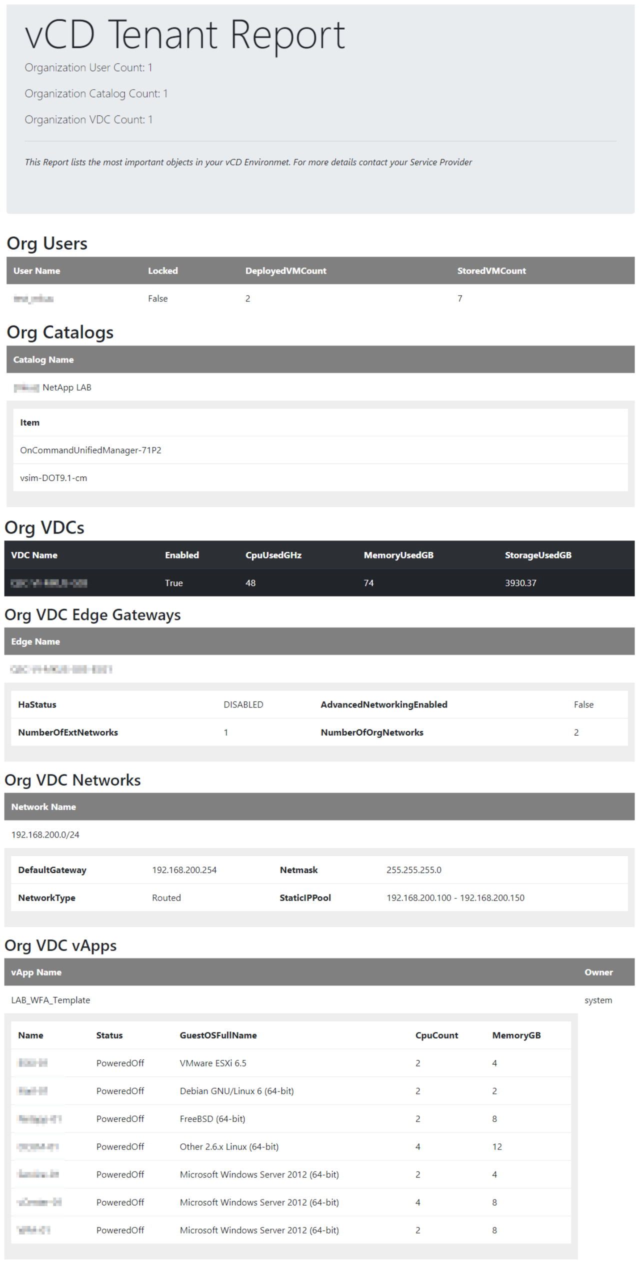
Markus Kraus
Follow on Twitter @vMarkus_K



Using PowerShell to Create a vCloud Director Tenant HTML Report

As we all know, VMware vCloud Director is a widely adopted IaaS platform for the service provider market. VMware vCloud Director offers a self-service web portal to manage your vApps, VMs, networks, and network functions (Edge Firewall, NAT, VPN, Load Balancer, DFW, and Rrouting). But there is also a RESTful API, and a PowerShell Module offered for the administrators and tenants. With the help of the API, some third– party vendors offer an extended web portal (for example, OnApp). In this article, I'll show you how to use the VMware vCloud Director PowerShell Module (part of the famous VMware PowerCLI) to extend the default UI with a vCloud Director Tenant HTML Report for your most important objects. Unfortunately, there is no reporting option offered by the self-service web portal itself.

The problem with extensive HTML reports created with PowerShell is that the ConvertTo-HTML Cmdlet is not really flexible. So I was looking for alternative ways and found the PowerShell module PowerStartHTML from Timothy Dewin. This module combines PowerShell with Bootstrap, a open source toolkit for developing with HTML, CSS, and JS. With this toolkit, I was able to create a report that contains the necessary information's in a nice–looking format.



vCD Tenant Report

Organization User Count: 1

Organization Catalog Count: 1

Organization VDC Count: 1

This Report lists the most important objects in your vCD Environmet. For more details contact your Service Provider

Org Users

User Name	Locked	DeployedVMCount	StoredVMCount
[redacted]	False	2	7

Org Catalogs

Catalog Name
[redacted] NetApp LAB

Item
OnCommandUnifiedManager-71P2
vsim-DOT9.1-cm

Org VDCs

VDC Name	Enabled	CpuUsedGHz	MemoryUsedGB	StorageUsedGB
[redacted]	True	48	74	3930.37

Org VDC Edge Gateways

Edge Name
[redacted]

HaStatus	DISABLED	AdvancedNetworkingEnabled	False
NumberOfExtNetworks	1	NumberOfOrgNetworks	2

Org VDC Networks

Network Name
192.168.200.0/24

DefaultGateway	192.168.200.254	Netmask	255.255.255.0
NetworkType	Routed	StaticIPPool	192.168.200.100 - 192.168.200.150

Org VDC vApps

vApp Name	Owner
LAB_WFA_Template	system

Name	Status	GuestOSFullName	CpuCount	MemoryGB
[redacted]	PoweredOff	VMware ESXi 6.5	2	4
[redacted]	PoweredOff	Debian GNU/Linux 6 (64-bit)	2	2
[redacted]	PoweredOff	FreeBSD (64-bit)	2	8
[redacted]	PoweredOff	Other 2.6.x Linux (64-bit)	4	12
[redacted]	PoweredOff	Microsoft Windows Server 2012 (64-bit)	2	4
[redacted]	PoweredOff	Microsoft Windows Server 2012 (64-bit)	4	8
[redacted]	PoweredOff	Microsoft Windows Server 2012 (64-bit)	2	8

VMWARE POWERCLI FOR VCLOUD DIRECTOR BASICS

The latest version of VMware PowerCLI is available on PowerShell Gallery. So if you use PowerShell 5.0 or newer, you can install PowerCLI modules with one simple command:

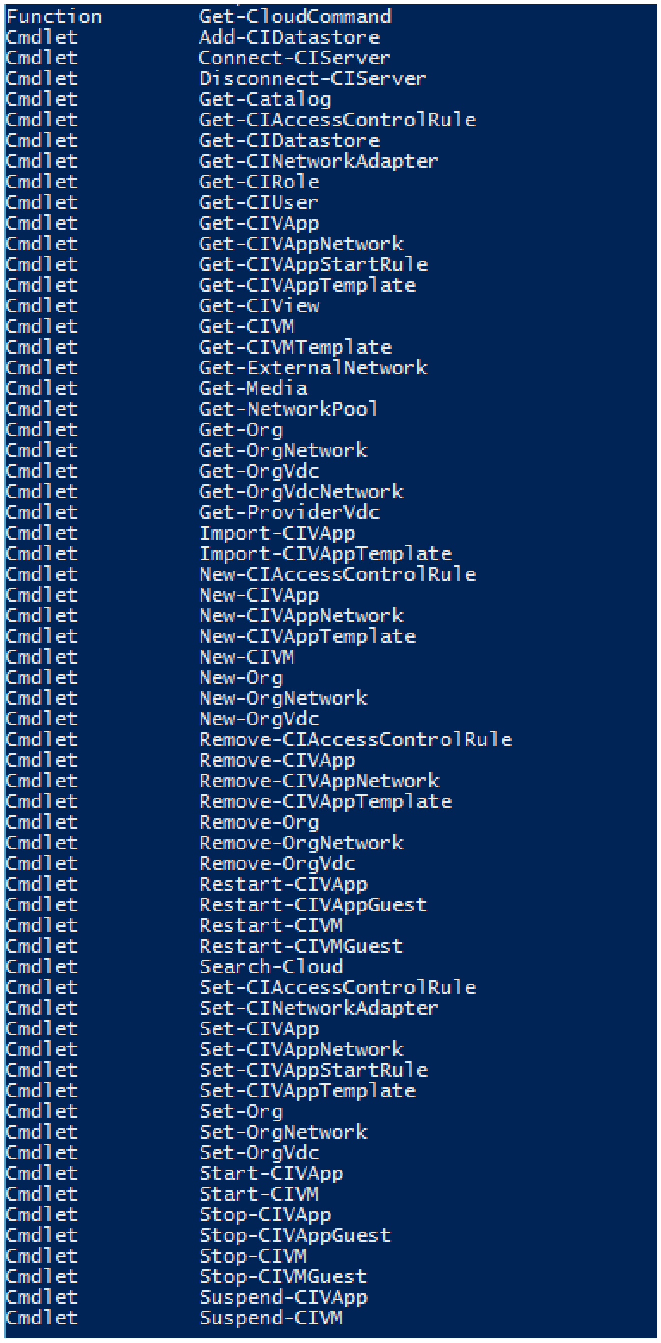
```
Install-Module VMware.PowerCLI -Scope CurrentUser
```

After installation is successful, you can load all VMware modules using this command:

```
Get-Module -Name VMware* -ListAvailable | Import-Module
```

One of the loaded modules is VMware.VimAutomation.Cloud, which is for VMware vCloud Director. But be aware that not all commands work as a tenant. Some Ccmdlets are for administrative use only and others are for vCloud Air (which was sold to OVH).

All available vCloud Director Cmdlets (and one function) on version 6.5.1:



```
Function        Get-CloudCommand
Cmdlet          Add-CIDatastore
Cmdlet          Connect-CIServer
Cmdlet          Disconnect-CIServer
Cmdlet          Get-Catalog
Cmdlet          Get-CIAccessControlRule
Cmdlet          Get-CIDatastore
Cmdlet          Get-CINetworkAdapter
Cmdlet          Get-CIRole
Cmdlet          Get-CIUser
Cmdlet          Get-CIVApp
Cmdlet          Get-CIVAppNetwork
Cmdlet          Get-CIVAppStartRule
Cmdlet          Get-CIVAppTemplate
Cmdlet          Get-CIView
Cmdlet          Get-CIVM
Cmdlet          Get-CIVMTemplate
Cmdlet          Get-ExternalNetwork
Cmdlet          Get-Media
Cmdlet          Get-NetworkPool
Cmdlet          Get-Org
Cmdlet          Get-OrgNetwork
Cmdlet          Get-OrgVdc
Cmdlet          Get-OrgVdcNetwork
Cmdlet          Get-ProviderVdc
Cmdlet          Import-CIVApp
Cmdlet          Import-CIVAppTemplate
Cmdlet          New-CIAccessControlRule
Cmdlet          New-CIVApp
Cmdlet          New-CIVAppNetwork
Cmdlet          New-CIVAppTemplate
Cmdlet          New-CIVM
Cmdlet          New-Org
Cmdlet          New-OrgNetwork
Cmdlet          New-OrgVdc
Cmdlet          Remove-CIAccessControlRule
Cmdlet          Remove-CIVApp
Cmdlet          Remove-CIVAppNetwork
Cmdlet          Remove-CIVAppTemplate
Cmdlet          Remove-Org
Cmdlet          Remove-OrgNetwork
Cmdlet          Remove-OrgVdc
Cmdlet          Restart-CIVApp
Cmdlet          Restart-CIVAppGuest
Cmdlet          Restart-CIVM
Cmdlet          Restart-CIVMGuest
Cmdlet          Search-Cloud
Cmdlet          Set-CIAccessControlRule
Cmdlet          Set-CINetworkAdapter
Cmdlet          Set-CIVApp
Cmdlet          Set-CIVAppNetwork
Cmdlet          Set-CIVAppStartRule
Cmdlet          Set-CIVAppTemplate
Cmdlet          Set-Org
Cmdlet          Set-OrgNetwork
Cmdlet          Set-OrgVdc
Cmdlet          Start-CIVApp
Cmdlet          Start-CIVM
Cmdlet          Stop-CIVApp
Cmdlet          Stop-CIVAppGuest
Cmdlet          Stop-CIVM
Cmdlet          Stop-CIVMGuest
Cmdlet          Suspend-CIVApp
Cmdlet          Suspend-CIVM
```

The aliases for vCloud Air are removed from this screenshot.

To connect to your VMware vCloud Director organization, the Cmdlet Connect-CIServer works as a provider administrator and also as a tenant. Tenants simply need to add their organization during connection:

```
Connect-CIServer -Server <vCD FQDN> -Org <Org Name> [-Credential <PS Credential>]
```

<u>Other PowerCLI vCloud Director examples:</u>

- PowerCLI vCloud Director Customer Provisioning
- PowerCLI – Create vCloud Director Edge Gateway
- vCloud Director Edge-Gateway IP Report

SOME INTERESTING CMDLETS

SEARCH-CLOUD

The Search-Cloud Cmdlet is the fastest way to get vCloud Director objects via PowerCLI. You can also list some objects for that since no typical Get-* command is available (for example, Edge Gateways).

<u>Example: List Edge Gateways:</u>

```
Search-Cloud -QueryType EdgeGateway
```

All QueryTypes can be found in the VMware vSphere 6.5 Documentation Center.

GET-CIVAPP

The Get-CIVApp Cmdlet lists all vCloud Director vApps. vApps are also known from VMware vSphere but are way more important in vCloud Director. A vApp can contain objects like VMs, networks, and network functions.

Example: List vApp ‚NetApp-HA‘:

```
Get-CIVApp -Name NetApp-HA
```

GET-CIVM

The Get-CIVM Cmdlet lists all vCloud Director VMs. VMs are in vCloud Director always children from vApps and also the Cmdlet can use Get-CIVApp as piped input.

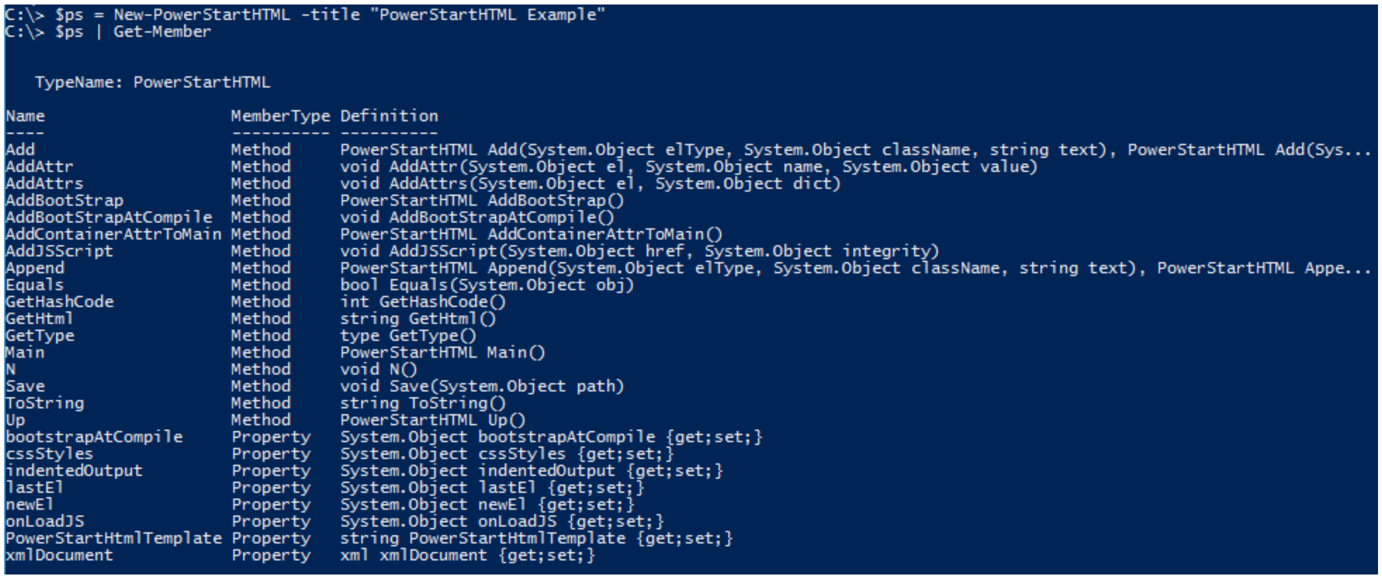
Example: List all VMs from vApp ‚NetApp-HA‘:

```
Get-CIVApp -Name NetApp-HA | Get-CIVM
```

BOOTSTRAP WITH POWERSHELL BASICS

Firs I have to say I am no HTML pro and I would not have been able to create the report without the help of this great example from Timothy Dewin. Timothy created an awesome Veeam Backup & Replication HTML Report with his PowerStartHTML PowerShell Module.

The function New-PowerStartHTML creates a new object with the necessary methods to create a full Bootstrap HTML page. The newly created object already includes the stylesheet for Bootstrap.



```
C:\> $ps = New-PowerStartHTML -title "PowerStartHTML Example"
C:\> $ps | Get-Member


   TypeName: PowerStartHTML

Name                    MemberType Definition
----                    ---------- ----------
Add                     Method     PowerStartHTML Add(System.Object elType, System.Object className, string text), PowerStartHTML Add(Sys...
AddAttr                 Method     void AddAttr(System.Object el, System.Object name, System.Object value)
AddAttrs                Method     void AddAttrs(System.Object el, System.Object dict)
AddBootStrap            Method     PowerStartHTML AddBootStrap()
AddBootStrapAtCompile   Method     void AddBootStrapAtCompile()
AddContainerAttrToMain  Method     PowerStartHTML AddContainerAttrToMain()
AddJSScript             Method     void AddJSScript(System.Object href, System.Object integrity)
Append                  Method     PowerStartHTML Append(System.Object elType, System.Object className, string text), PowerStartHTML Appe...
Equals                  Method     bool Equals(System.Object obj)
GetHashCode             Method     int GetHashCode()
GetHtml                 Method     string GetHtml()
GetType                 Method     type GetType()
Main                    Method     PowerStartHTML Main()
N                       Method     void N()
Save                    Method     void Save(System.Object path)
ToString                Method     string ToString()
Up                      Method     PowerStartHTML Up()
bootstrapAtCompile      Property   System.Object bootstrapAtCompile {get;set;}
cssStyles               Property   System.Object cssStyles {get;set;}
indentedOutput          Property   System.Object indentedOutput {get;set;}
lastEl                  Property   System.Object lastEl {get;set;}
newEl                   Property   System.Object newEl {get;set;}
onLoadJS                Property   System.Object onLoadJS {get;set;}
PowerStartHtmlTemplate  Property   string PowerStartHtmlTemplate {get;set;}
xmlDocument             Property   xml xmlDocument {get;set;}
```

The two main methods to create your HTML page are:

```
$ps.Add(type,class,text)
$ps.Append(type,class,text)
```

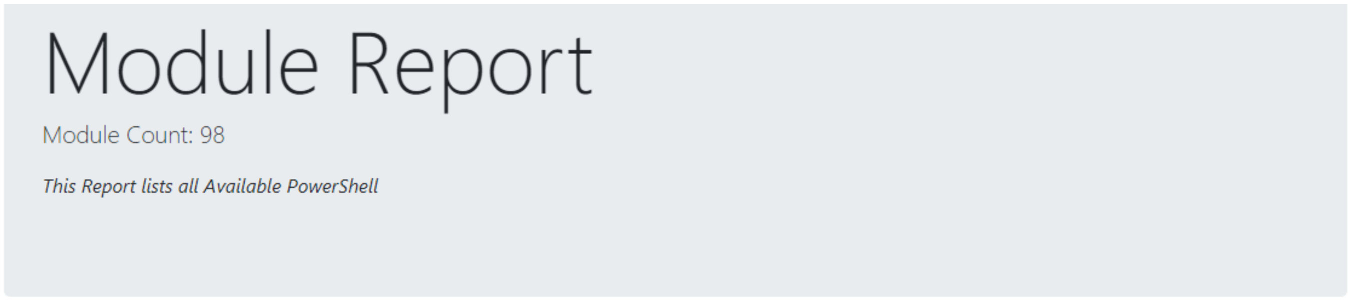
Here's an explanation from the README on GitHub:

"The difference between Add is that it will remember the new element as the as the element. With Append, it remember the parent, so you can add an element at the same depth.

All the possible types and classes can be found in the Bootstrap documentation.

To get started with a simple HTML file, I created a report for all available modules on my PC. The main part of the report is a simple table.

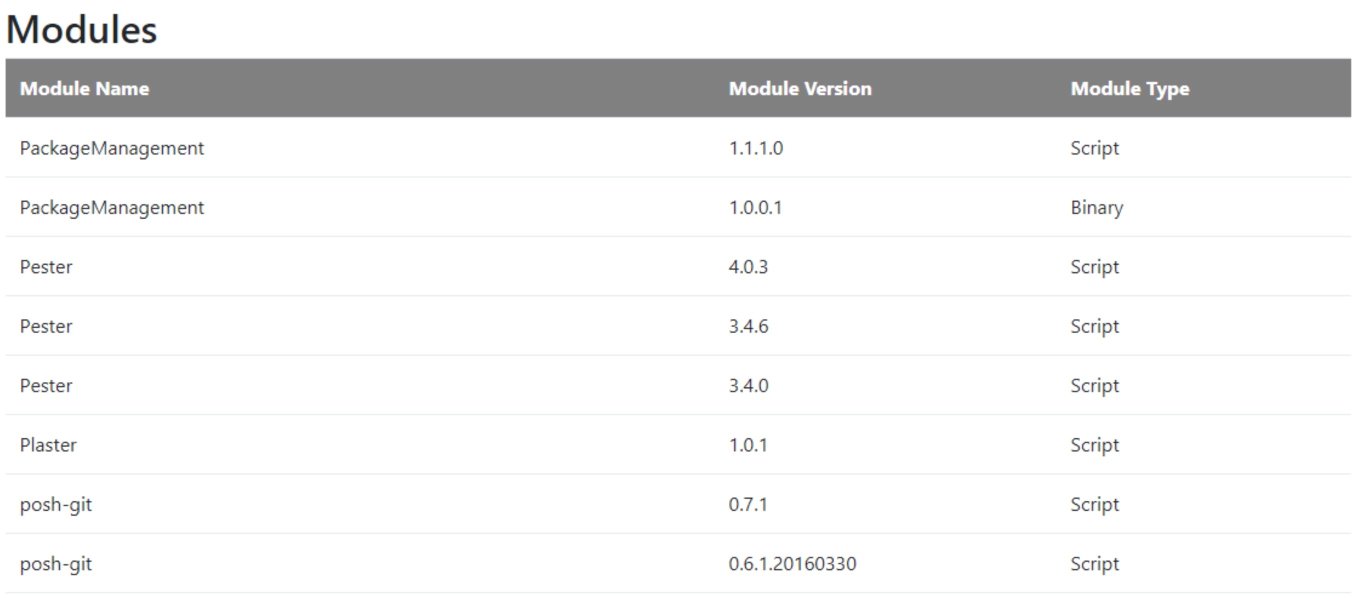
```
$Modules = Get-Module -ListAvailable
$Path = "C:\Temp\Example.html"
$ps = New-PowerStartHTML -title "PowerStartHTML Example"
$ps.cssStyles['.bgtitle'] = "background-color:grey"
$ps.Main().Add("div","jumbotron").N()
$ps.Append("h1","display-3",("Module Report")).Append("p","lead","Module
Count: {0}" -f $Modules.Count).Append("p","font-italic","This Report lists all
Available PowerShell").N()
$ps.Main().Append("h2",$null,"Modules").N()
$ps.Add('table','table').Add("tr","bgtitle textwhite").Append("th",$null,"Module Name").Append("th",$null,"Module
Version").Append("th",$null,"Module Type").N()
foreach ($Module in $Modules) {
$ps.Add("tr").N()
$ps.Append("td",$null,$Module.Name).N()
$ps.Append("td",$null,$Module.Version).N()
$ps.Append("td",$null,$Module.ModuleType).N()
$ps.Up().N()
}$
ps.Up().N()
$ps.save($Path)
Start-Process $Path
```



Module Report

Module Count: 98

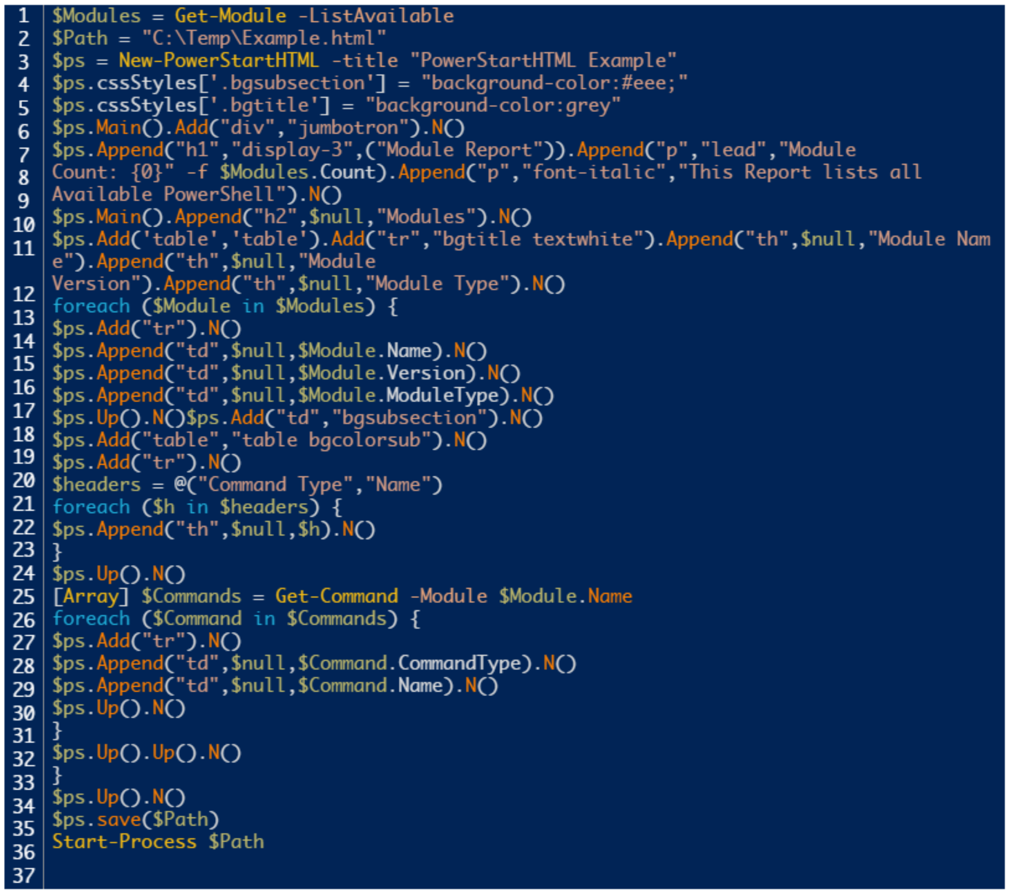
This Report lists all Available PowerShell



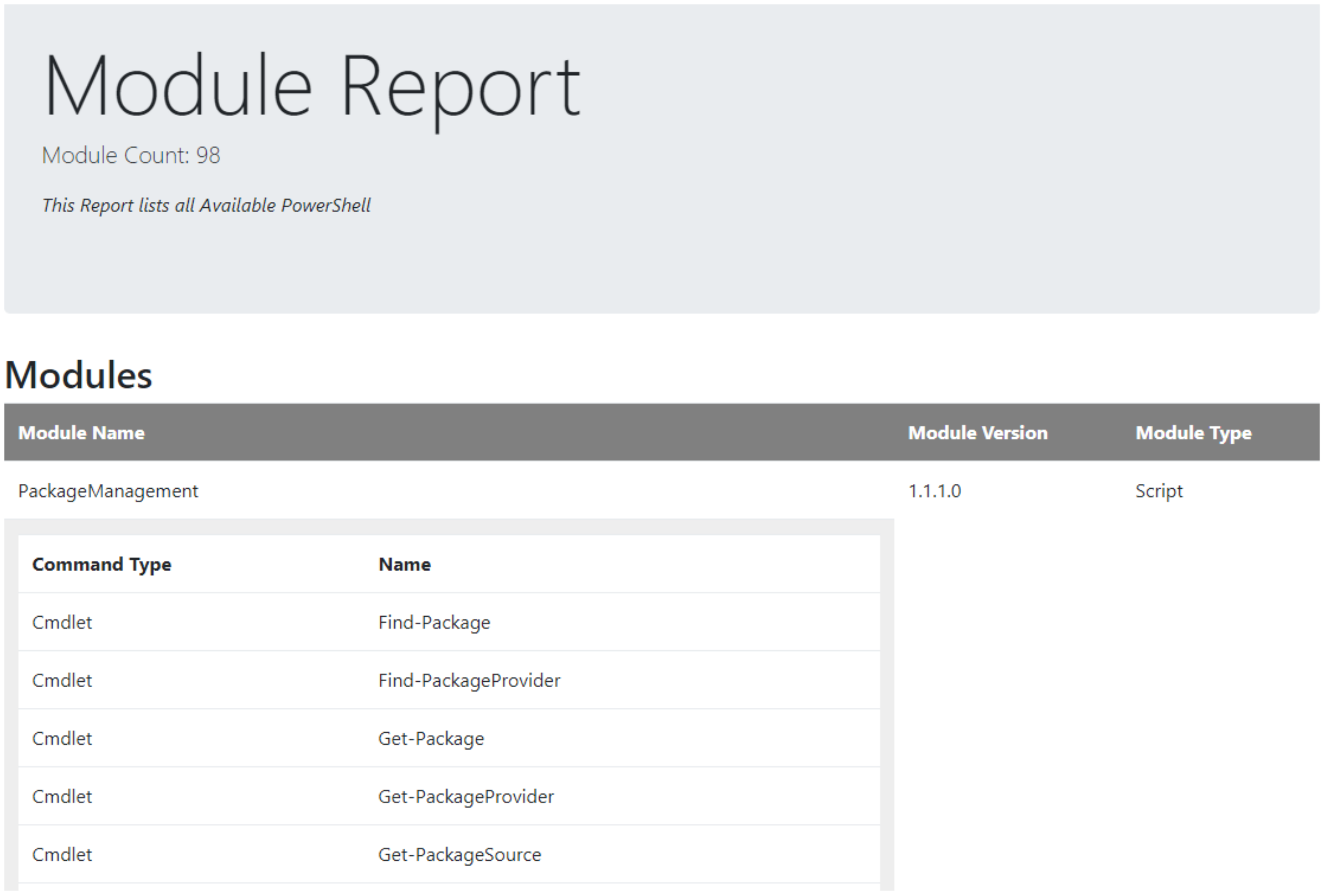
Modules

Module Name	Module Version	Module Type
PackageManagement	1.1.1.0	Script
PackageManagement	1.0.0.1	Binary
Pester	4.0.3	Script
Pester	3.4.6	Script
Pester	3.4.0	Script
Plaster	1.0.1	Script
posh-git	0.7.1	Script
posh-git	0.6.1.20160330	Script

This Eexample is not very complex and it maybe also be possible to create a simple table like this with the ConvertTo-HTML Cmdlet. But let's get a little bit more complex and add the commands of each Module in a sub-section of the table.



```
$Modules = Get-Module -ListAvailable
$Path = "C:\Temp\Example.html"
$ps = New-PowerStartHTML -title "PowerStartHTML Example"
$ps.cssStyles['.bgsubsection'] = "background-color:#eee;"
$ps.cssStyles['.bgtitle'] = "background-color:grey"
$ps.Main().Add("div","jumbotron").N()
$ps.Append("h1","display-3",("Module Report")).Append("p","lead","Module
Count: {0}" -f $Modules.Count).Append("p","font-italic","This Report lists all
Available PowerShell").N()
$ps.Main().Append("h2",$null,"Modules").N()
$ps.Add('table','table').Add("tr","bgtitle textwhite").Append("th",$null,"Module Nam
e").Append("th",$null,"Module
Version").Append("th",$null,"Module Type").N()
foreach ($Module in $Modules) {
$ps.Add("tr").N()
$ps.Append("td",$null,$Module.Name).N()
$ps.Append("td",$null,$Module.Version).N()
$ps.Append("td",$null,$Module.ModuleType).N()
$ps.Up().N()$ps.Add("td","bgsubsection").N()
$ps.Add("table","table bgcolorsub").N()
$ps.Add("tr").N()
$headers = @("Command Type","Name")
foreach ($h in $headers) {
$ps.Append("th",$null,$h).N()
}
$ps.Up().N()
[Array] $Commands = Get-Command -Module $Module.Name
foreach ($Command in $Commands) {
$ps.Add("tr").N()
$ps.Append("td",$null,$Command.CommandType).N()
$ps.Append("td",$null,$Command.Name).N()
$ps.Up().N()
}
$ps.Up().Up().N()
}
$ps.Up().N()
$ps.save($Path)
Start-Process $Path
```



Module Report

Module Count: 98

This Report lists all Available PowerShell

Modules

Module Name	**Module Version**	**Module Type**
PackageManagement	1.1.1.0	Script

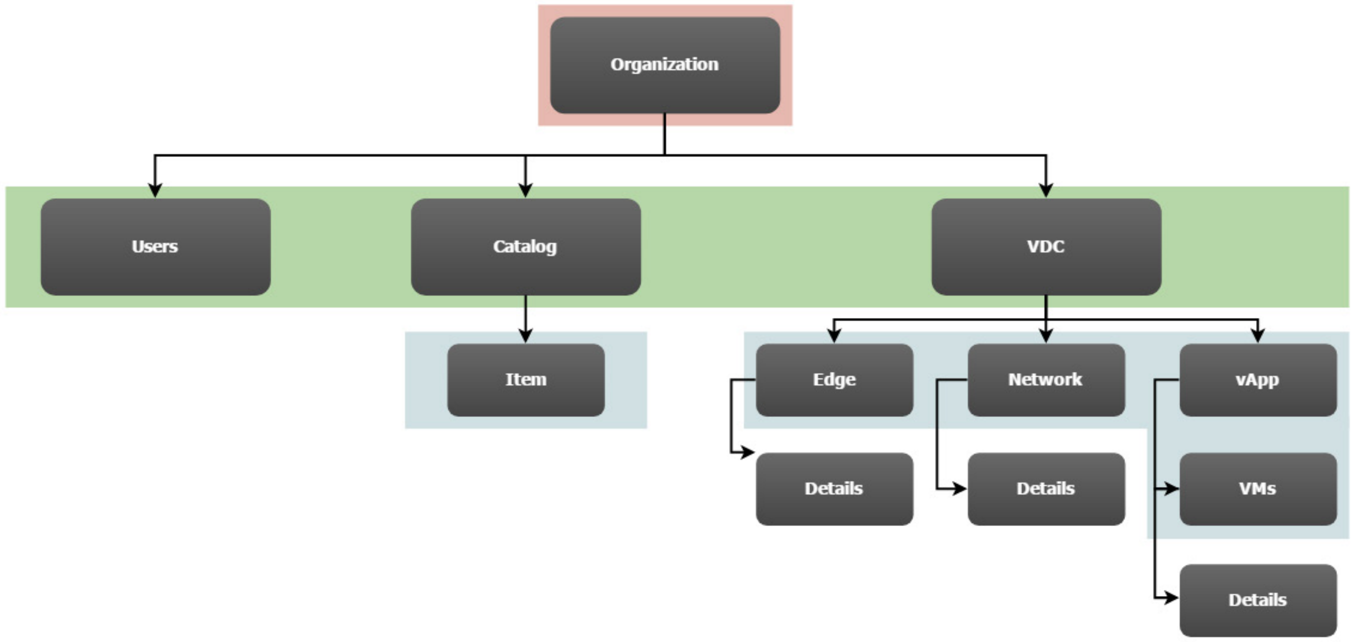
Command Type	**Name**
Cmdlet	Find-Package
Cmdlet	Find-PackageProvider
Cmdlet	Get-Package
Cmdlet	Get-PackageProvider
Cmdlet	Get-PackageSource

Okay, so that table is definitely more complex than what you can create with the ConvertTo-HTML Ccmdlet!

THE VCLOUD DIRECTOR TENANT HTML REPORT

Now that we're familiar with the VMware PowerCLI vCloud Director Module and the basics of Bootstrap PowerShell handling, let`s get started with the report.

The first step is to decide which data is necessary for the vCloud Director Tenant HTML Report. This is my selection:



With this set of data, we are now able to collect the object details from our VMware vCloud Director server using VMware PowerCLI.

OBJECT DETAILS

GET USERS

```
[Array] $Users = Get-CIUser
```

GET CATALOGS

```
[Array] $Catalogs = Get-Catalog
```

GET CATALOG ITEMS

```
[Array] $Items = $Catalog.ExtensionData.CatalogItems.CatalogItem
```

GET VDCS

```
[Array] $OrgVdcs = Get-OrgVdc
```

GET VDC EDGE GATEWAYS

```
[Array] $Edges = Search-Cloud -QueryType EdgeGateway -Filter "Vdc==$($OrgVdc.Id)"
```

GET VDC NETWORKS

```powershell
[Array] $Networks = $OrgVdc | Get-OrgVdcNetwork
```

GET VDC VAPPS

```powershell
[Array] $Vapps = $OrgVdc | Get-CIVApp
```

GET VDC VAPP VMS

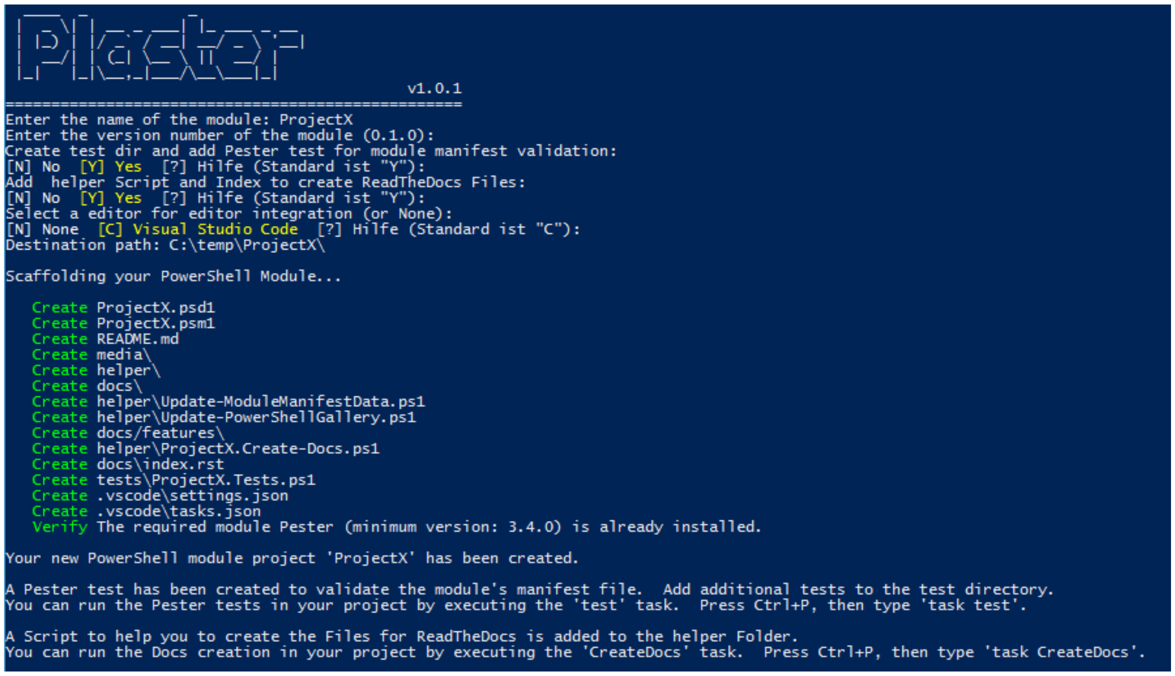
```powershell
[Array] $VMs = $Vapp | Get-CIVM
```

FINAL VCLOUD DIRECTOR TENANT HTML REPORT

I wanted to ship the vCloud Director Tenant HTML Report script as a PowerShell Module since that is the best way to also include the PowerStartHTML Module.

HOW TO CREATE A MODULE

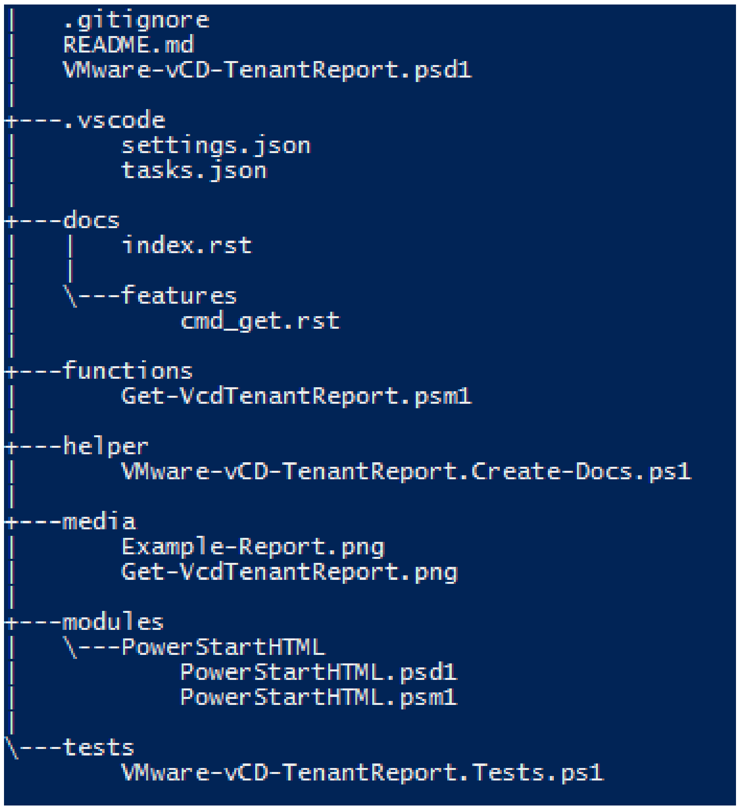
To create my PowerShell modules in a standardized way I use the Plaster Module with a customized template. For more details about Plaster and Plaster templates, you can read my blog article.



```
  ____  _           _
 |  _ \| | __ _ ___| |_ ___ _ __
 | |_) | |/ _` / __| __/ _ \ '__|
 |  __/| | (_| \__ \ ||  __/ |
 |_|   |_|\__,_|___/\__\___|_|
                                             v1.0.1
==================================================
Enter the name of the module: ProjectX
Enter the version number of the module (0.1.0):
Create test dir and add Pester test for module manifest validation:
[N] No  [Y] Yes  [?] Hilfe (Standard ist "Y"):
Add  helper Script and Index to create ReadTheDocs Files:
[N] No  [Y] Yes  [?] Hilfe (Standard ist "Y"):
Select a editor for editor integration (or None):
[N] None  [C] Visual Studio Code  [?] Hilfe (Standard ist "C"):
Destination path: C:\temp\ProjectX\

Scaffolding your PowerShell Module...

   Create ProjectX.psd1
   Create ProjectX.psm1
   Create README.md
   Create media\
   Create helper\
   Create docs\
   Create helper\Update-ModuleManifestData.ps1
   Create helper\Update-PowerShellGallery.ps1
   Create docs/features\
   Create helper\ProjectX.Create-Docs.ps1
   Create docs\index.rst
   Create tests\ProjectX.Tests.ps1
   Create .vscode\settings.json
   Create .vscode\tasks.json
   Verify The required module Pester (minimum version: 3.4.0) is already installed.

Your new PowerShell module project 'ProjectX' has been created.

A Pester test has been created to validate the module's manifest file.  Add additional tests to the test directory.
You can run the Pester tests in your project by executing the 'test' task.  Press Ctrl+P, then type 'task test'.

A Script to help you to create the Files for ReadTheDocs is added to the helper Folder.
You can run the Docs creation in your project by executing the 'CreateDocs' task.  Press Ctrl+P, then type 'task CreateDocs'.
```

THE ‚VMWARE-VCD-TENANTREPORT' MODULE

With my Plaster template and some manual modifications, I have created a Module with PowerStartHTML as a Sub-Module. The module also ships with a basic Pester test.

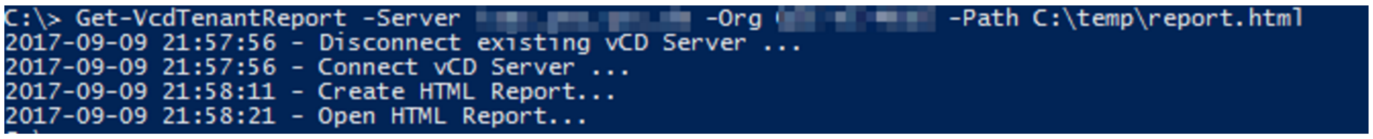


```
|   .gitignore
|   README.md
|   VMware-vCD-TenantReport.psd1
|
+---.vscode
|       settings.json
|       tasks.json
|
+---docs
|   |   index.rst
|   |
|   \---features
|           cmd_get.rst
|
+---functions
|       Get-VcdTenantReport.psm1
|
+---helper
|       VMware-vCD-TenantReport.Create-Docs.ps1
|
+---media
|       Example-Report.png
|       Get-VcdTenantReport.png
|
+---modules
|   \---PowerStartHTML
|           PowerStartHTML.psd1
|           PowerStartHTML.psm1
|
\---tests
        VMware-vCD-TenantReport.Tests.ps1
```

THE REPORT SCRIPT

The script to create the vCloud Director Tenant HTML Report is embedded in the exported Get-VcdTenantReport function, The usage is quite simple:

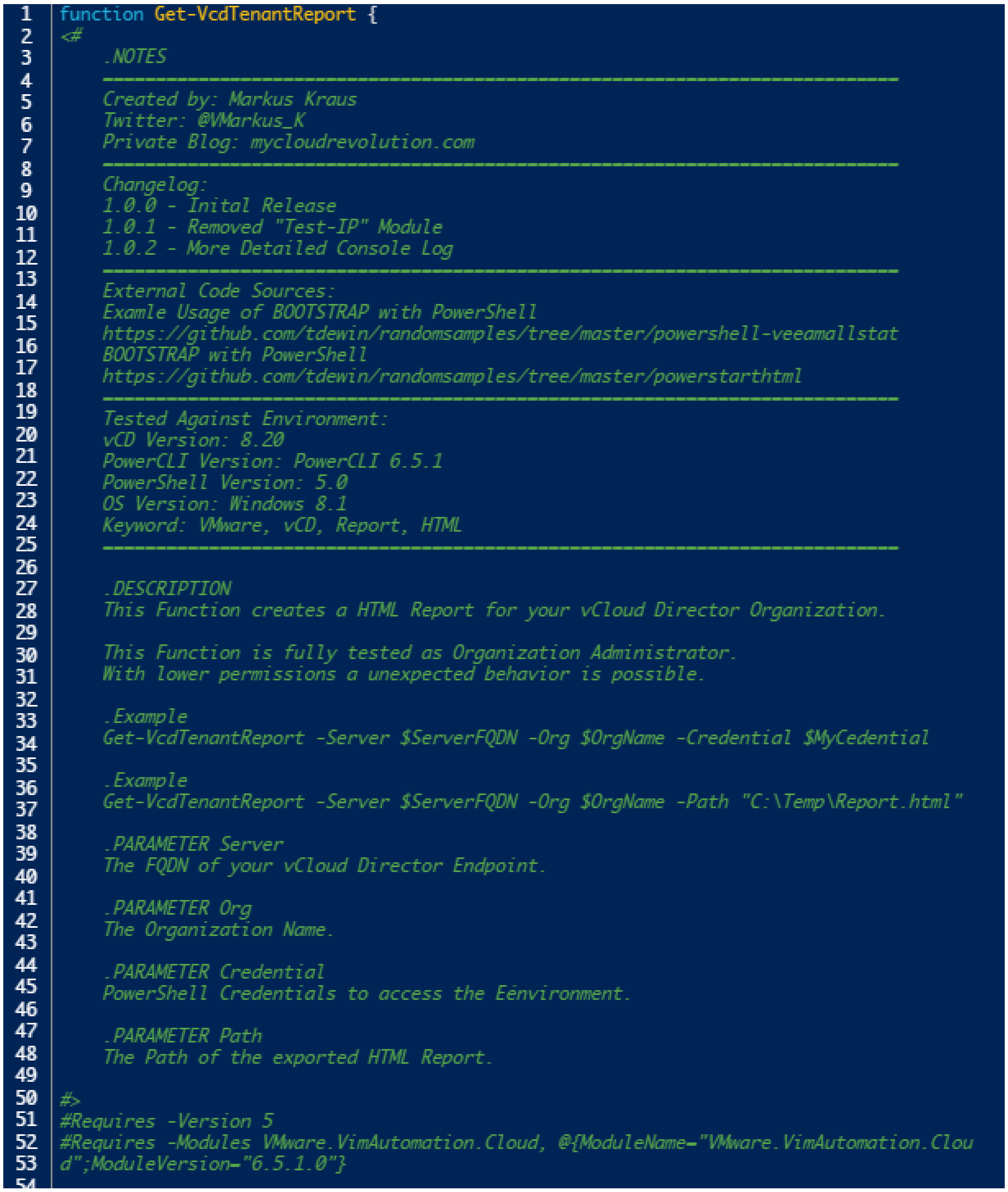
```
Get-VcdTenantReport -Server <vCD FQDN> -Org <Org Name> [-Credential $PSCred -Path
C:\temp\report.html]
```



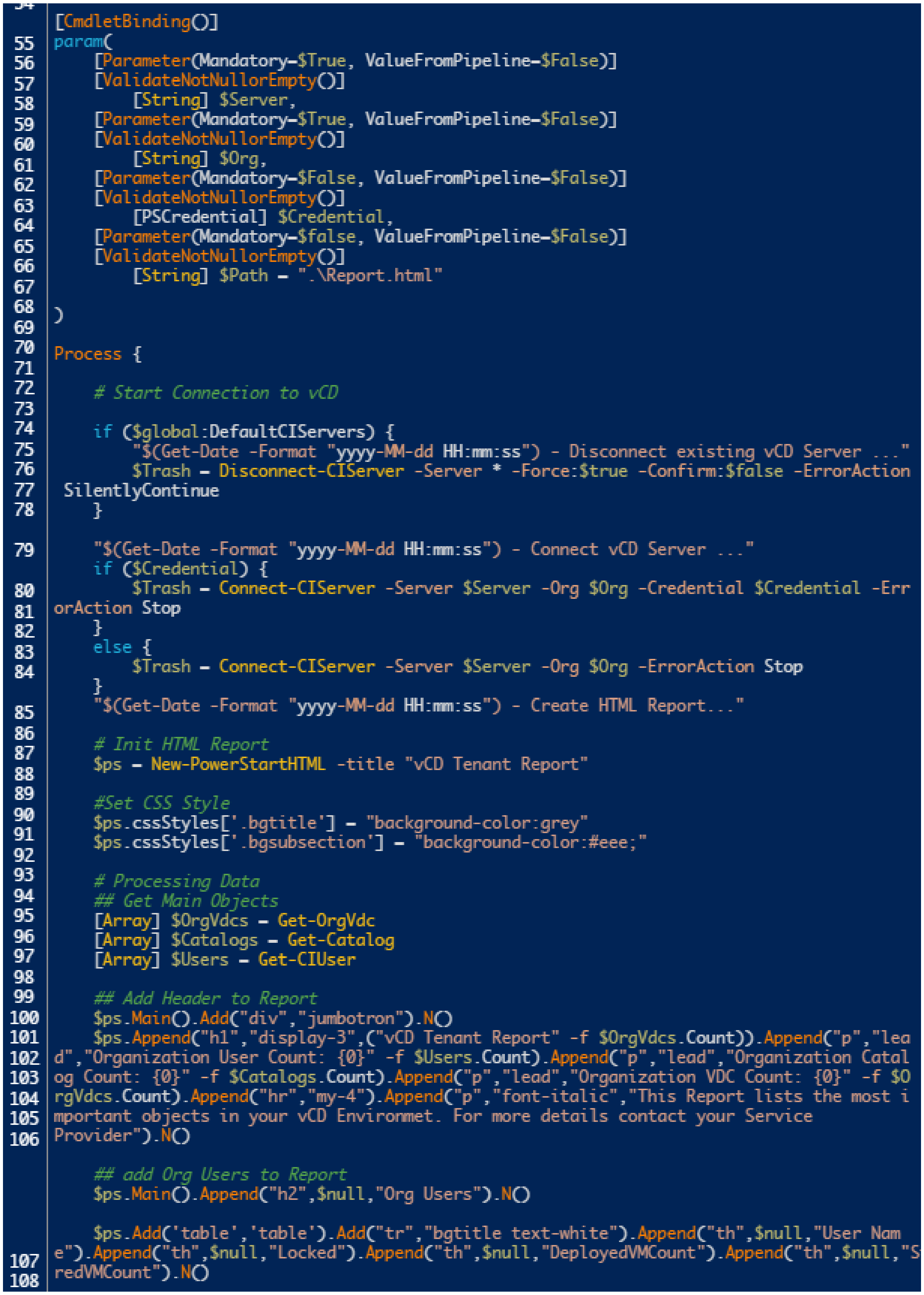
```
C:\> Get-VcdTenantReport -Server [redacted] -Org [redacted] -Path C:\temp\report.html
2017-09-09 21:57:56 - Disconnect existing vCD Server ...
2017-09-09 21:57:56 - Connect vCD Server ...
2017-09-09 21:58:11 - Create HTML Report...
2017-09-09 21:58:21 - Open HTML Report...
```

The report will be automatically opened in your default browser.

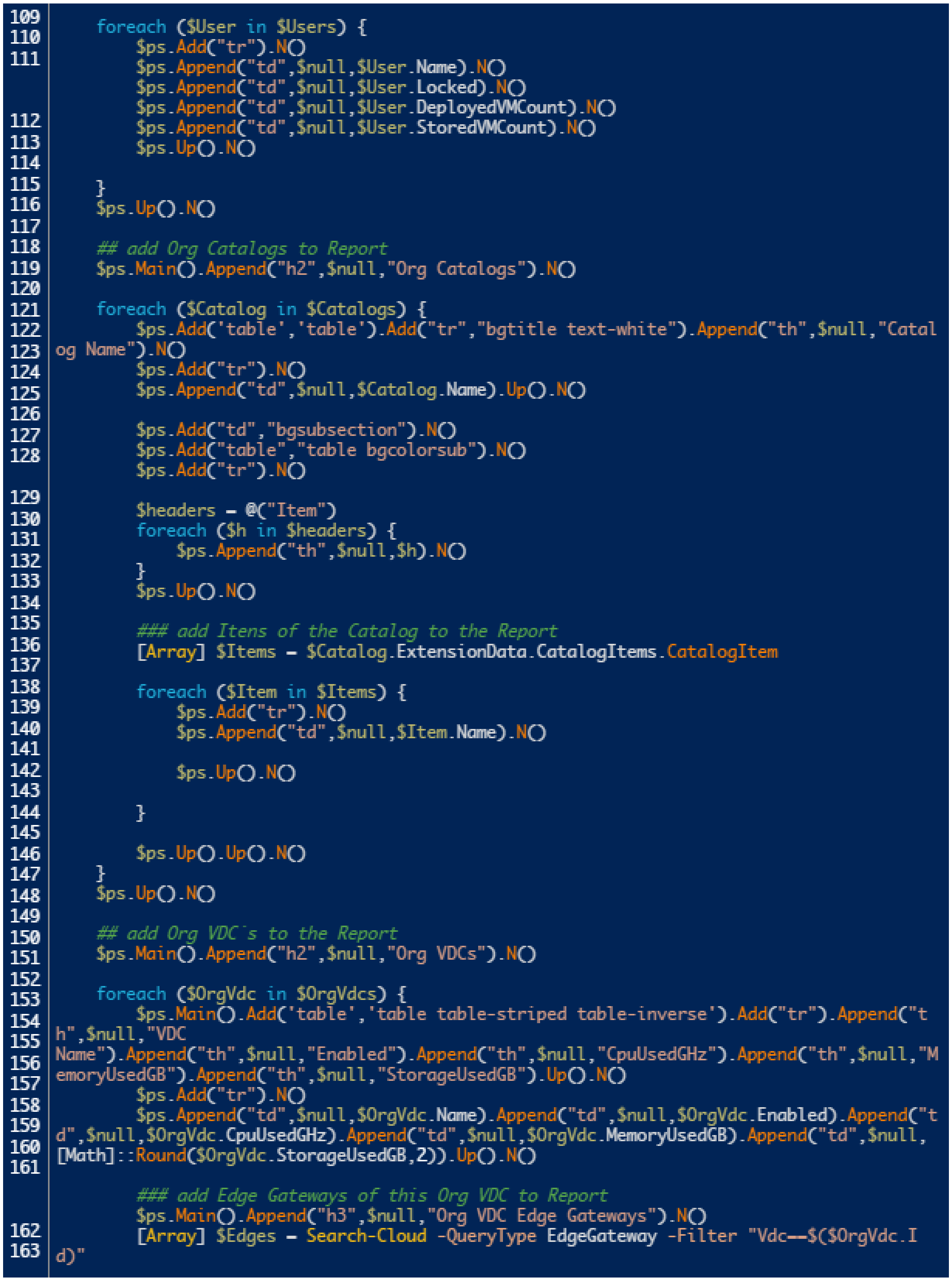
VCLOUD DIRECTOR TENANT HTML REPORT POWERSCODE



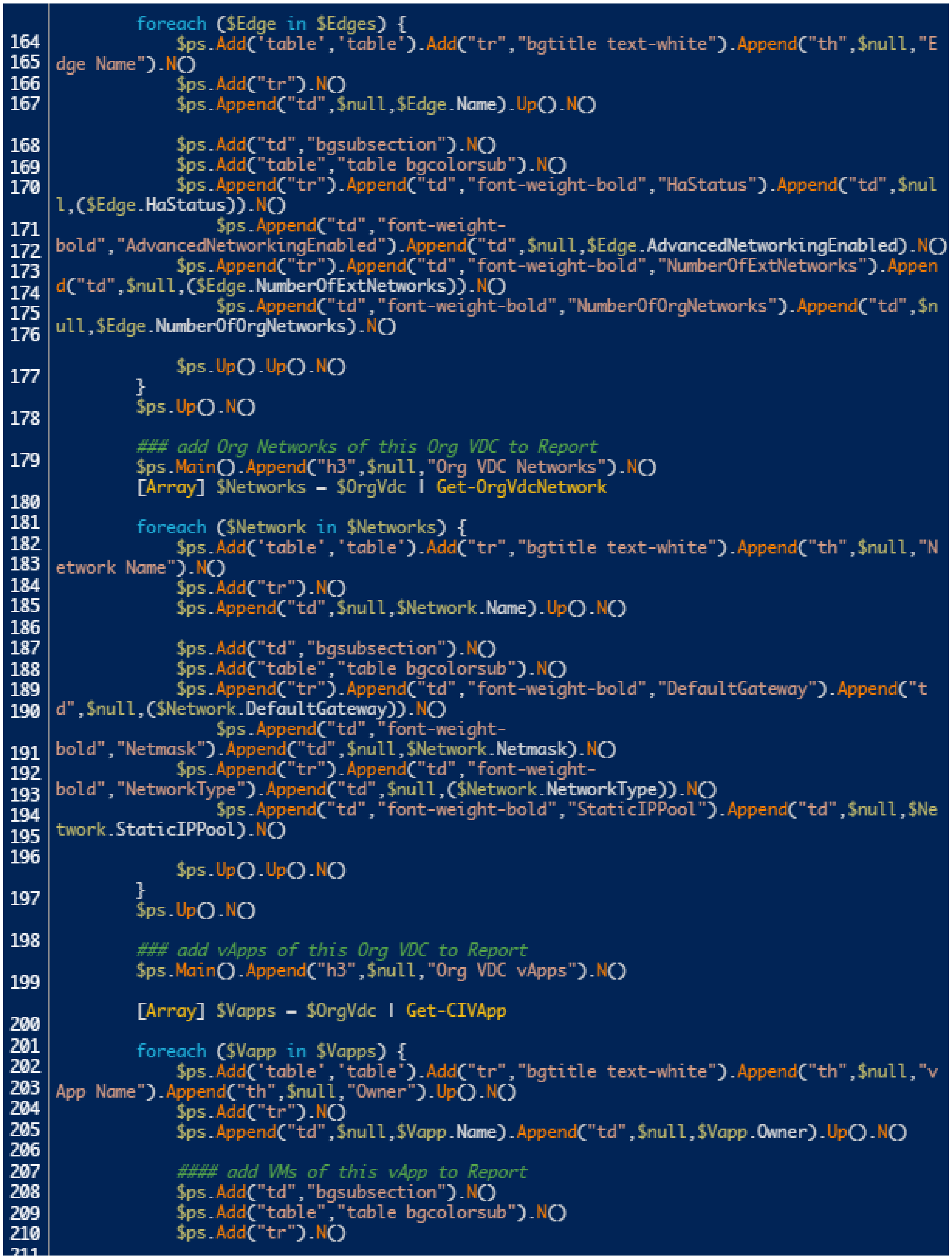
```
function Get-VcdTenantReport {
<#
    .NOTES
    ===========================================================================
    Created by: Markus Kraus
    Twitter: @VMarkus_K
    Private Blog: mycloudrevolution.com
    ===========================================================================
    Changelog:
    1.0.0 - Inital Release
    1.0.1 - Removed "Test-IP" Module
    1.0.2 - More Detailed Console Log
    ===========================================================================
    External Code Sources:
    Examle Usage of BOOTSTRAP with PowerShell
    https://github.com/tdewin/randomsamples/tree/master/powershell-veeamallstat
    BOOTSTRAP with PowerShell
    https://github.com/tdewin/randomsamples/tree/master/powerstarthtml
    ===========================================================================
    Tested Against Environment:
    vCD Version: 8.20
    PowerCLI Version: PowerCLI 6.5.1
    PowerShell Version: 5.0
    OS Version: Windows 8.1
    Keyword: VMware, vCD, Report, HTML
    ===========================================================================

    .DESCRIPTION
    This Function creates a HTML Report for your vCloud Director Organization.

    This Function is fully tested as Organization Administrator.
    With lower permissions a unexpected behavior is possible.

    .Example
    Get-VcdTenantReport -Server $ServerFQDN -Org $OrgName -Credential $MyCedential

    .Example
    Get-VcdTenantReport -Server $ServerFQDN -Org $OrgName -Path "C:\Temp\Report.html"

    .PARAMETER Server
    The FQDN of your vCloud Director Endpoint.

    .PARAMETER Org
    The Organization Name.

    .PARAMETER Credential
    PowerShell Credentials to access the Eenvironment.

    .PARAMETER Path
    The Path of the exported HTML Report.

#>
#Requires -Version 5
#Requires -Modules VMware.VimAutomation.Cloud, @{ModuleName="VMware.VimAutomation.Clou
d";ModuleVersion="6.5.1.0"}
```



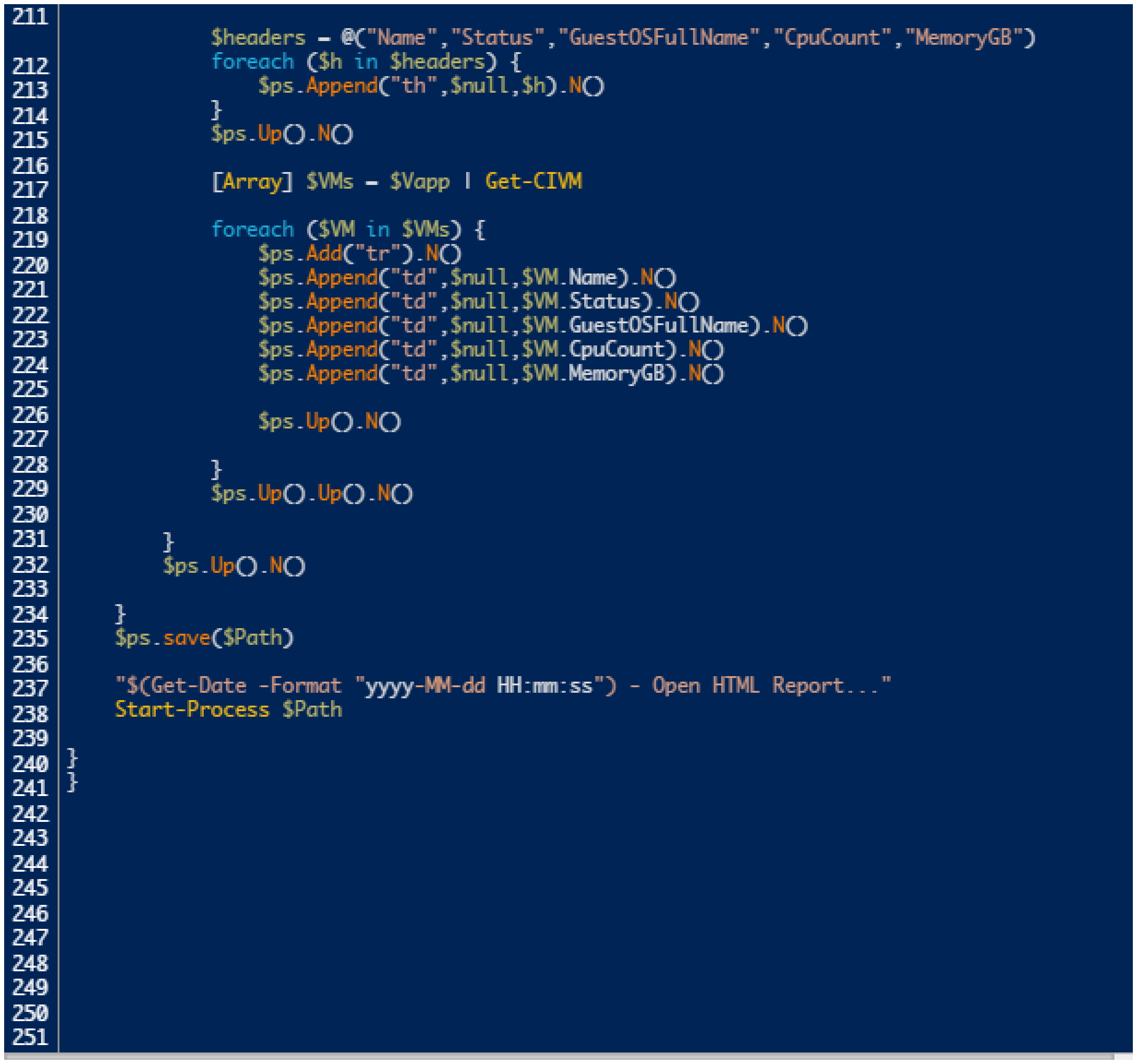
```
[CmdletBinding()]
param(
    [Parameter(Mandatory=$True, ValueFromPipeline=$False)]
    [ValidateNotNullorEmpty()]
        [String] $Server,
    [Parameter(Mandatory=$True, ValueFromPipeline=$False)]
    [ValidateNotNullorEmpty()]
        [String] $Org,
    [Parameter(Mandatory=$False, ValueFromPipeline=$False)]
    [ValidateNotNullorEmpty()]
        [PSCredential] $Credential,
    [Parameter(Mandatory=$false, ValueFromPipeline=$False)]
    [ValidateNotNullorEmpty()]
        [String] $Path = ".\Report.html"

)

Process {

    # Start Connection to vCD

    if ($global:DefaultCIServers) {
        "$(Get-Date -Format "yyyy-MM-dd HH:mm:ss") - Disconnect existing vCD Server ..."
        $Trash = Disconnect-CIServer -Server * -Force:$true -Confirm:$false -ErrorAction
 SilentlyContinue
    }

    "$(Get-Date -Format "yyyy-MM-dd HH:mm:ss") - Connect vCD Server ..."
    if ($Credential) {
        $Trash = Connect-CIServer -Server $Server -Org $Org -Credential $Credential -Err
orAction Stop
    }
    else {
        $Trash = Connect-CIServer -Server $Server -Org $Org -ErrorAction Stop
    }
    "$(Get-Date -Format "yyyy-MM-dd HH:mm:ss") - Create HTML Report..."

    # Init HTML Report
    $ps = New-PowerStartHTML -title "vCD Tenant Report"

    #Set CSS Style
    $ps.cssStyles['.bgtitle'] = "background-color:grey"
    $ps.cssStyles['.bgsubsection'] = "background-color:#eee;"

    # Processing Data
    ## Get Main Objects
    [Array] $OrgVdcs = Get-OrgVdc
    [Array] $Catalogs = Get-Catalog
    [Array] $Users = Get-CIUser

    ## Add Header to Report
    $ps.Main().Add("div","jumbotron").N()
    $ps.Append("h1","display-3",("vCD Tenant Report" -f $OrgVdcs.Count)).Append("p","lea
d","Organization User Count: {0}" -f $Users.Count).Append("p","lead","Organization Catal
og Count: {0}" -f $Catalogs.Count).Append("p","lead","Organization VDC Count: {0}" -f $O
rgVdcs.Count).Append("hr","my-4").Append("p","font-italic","This Report lists the most i
mportant objects in your vCD Environmet. For more details contact your Service
Provider").N()

    ## add Org Users to Report
    $ps.Main().Append("h2",$null,"Org Users").N()

    $ps.Add('table','table').Add("tr","bgtitle text-white").Append("th",$null,"User Nam
e").Append("th",$null,"Locked").Append("th",$null,"DeployedVMCount").Append("th",$null,"S
redVMCount").N()
```



```
    foreach ($User in $Users) {
        $ps.Add("tr").N()
        $ps.Append("td",$null,$User.Name).N()
        $ps.Append("td",$null,$User.Locked).N()
        $ps.Append("td",$null,$User.DeployedVMCount).N()
        $ps.Append("td",$null,$User.StoredVMCount).N()
        $ps.Up().N()

    }
    $ps.Up().N()

    ## add Org Catalogs to Report
    $ps.Main().Append("h2",$null,"Org Catalogs").N()

    foreach ($Catalog in $Catalogs) {
        $ps.Add('table','table').Add("tr","bgtitle text-white").Append("th",$null,"Catalog Name").N()
        $ps.Add("tr").N()
        $ps.Append("td",$null,$Catalog.Name).Up().N()

        $ps.Add("td","bgsubsection").N()
        $ps.Add("table","table bgcolorsub").N()
        $ps.Add("tr").N()

        $headers = @("Item")
        foreach ($h in $headers) {
            $ps.Append("th",$null,$h).N()
        }
        $ps.Up().N()

        ### add Itens of the Catalog to the Report
        [Array] $Items = $Catalog.ExtensionData.CatalogItems.CatalogItem

        foreach ($Item in $Items) {
            $ps.Add("tr").N()
            $ps.Append("td",$null,$Item.Name).N()

            $ps.Up().N()

        }

        $ps.Up().Up().N()
    }
    $ps.Up().N()

    ## add Org VDC`s to the Report
    $ps.Main().Append("h2",$null,"Org VDCs").N()

    foreach ($OrgVdc in $OrgVdcs) {
        $ps.Main().Add('table','table table-striped table-inverse').Add("tr").Append("th",$null,"VDC
Name").Append("th",$null,"Enabled").Append("th",$null,"CpuUsedGHz").Append("th",$null,"MemoryUsedGB").Append("th",$null,"StorageUsedGB").Up().N()
        $ps.Add("tr").N()
        $ps.Append("td",$null,$OrgVdc.Name).Append("td",$null,$OrgVdc.Enabled).Append("td",$null,$OrgVdc.CpuUsedGHz).Append("td",$null,$OrgVdc.MemoryUsedGB).Append("td",$null,
[Math]::Round($OrgVdc.StorageUsedGB,2)).Up().N()

        ### add Edge Gateways of this Org VDC to Report
        $ps.Main().Append("h3",$null,"Org VDC Edge Gateways").N()
        [Array] $Edges = Search-Cloud -QueryType EdgeGateway -Filter "Vdc==$($OrgVdc.Id)"
```



```
        foreach ($Edge in $Edges) {
            $ps.Add('table','table').Add("tr","bgtitle text-white").Append("th",$null,"E
dge Name").N()
            $ps.Add("tr").N()
            $ps.Append("td",$null,$Edge.Name).Up().N()

            $ps.Add("td","bgsubsection").N()
            $ps.Add("table","table bgcolorsub").N()
            $ps.Append("tr").Append("td","font-weight-bold","HaStatus").Append("td",$nul
l,($Edge.HaStatus)).N()
                $ps.Append("td","font-weight-
bold","AdvancedNetworkingEnabled").Append("td",$null,$Edge.AdvancedNetworkingEnabled).N()
            $ps.Append("tr").Append("td","font-weight-bold","NumberOfExtNetworks").Appen
d("td",$null,($Edge.NumberOfExtNetworks)).N()
                $ps.Append("td","font-weight-bold","NumberOfOrgNetworks").Append("td",$n
ull,$Edge.NumberOfOrgNetworks).N()

            $ps.Up().Up().N()
        }
        $ps.Up().N()

        ### add Org Networks of this Org VDC to Report
        $ps.Main().Append("h3",$null,"Org VDC Networks").N()
        [Array] $Networks = $OrgVdc | Get-OrgVdcNetwork

        foreach ($Network in $Networks) {
            $ps.Add('table','table').Add("tr","bgtitle text-white").Append("th",$null,"N
etwork Name").N()
            $ps.Add("tr").N()
            $ps.Append("td",$null,$Network.Name).Up().N()

            $ps.Add("td","bgsubsection").N()
            $ps.Add("table","table bgcolorsub").N()
            $ps.Append("tr").Append("td","font-weight-bold","DefaultGateway").Append("t
d",$null,($Network.DefaultGateway)).N()
                $ps.Append("td","font-weight-
bold","Netmask").Append("td",$null,$Network.Netmask).N()
            $ps.Append("tr").Append("td","font-weight-
bold","NetworkType").Append("td",$null,($Network.NetworkType)).N()
                $ps.Append("td","font-weight-bold","StaticIPPool").Append("td",$null,$Ne
twork.StaticIPPool).N()

            $ps.Up().Up().N()
        }
        $ps.Up().N()

        ### add vApps of this Org VDC to Report
        $ps.Main().Append("h3",$null,"Org VDC vApps").N()

        [Array] $Vapps = $OrgVdc | Get-CIVApp

        foreach ($Vapp in $Vapps) {
            $ps.Add('table','table').Add("tr","bgtitle text-white").Append("th",$null,"v
App Name").Append("th",$null,"Owner").Up().N()
            $ps.Add("tr").N()
            $ps.Append("td",$null,$Vapp.Name).Append("td",$null,$Vapp.Owner).Up().N()

            #### add VMs of this vApp to Report
            $ps.Add("td","bgsubsection").N()
            $ps.Add("table","table bgcolorsub").N()
            $ps.Add("tr").N()
```



```
            $headers = @("Name","Status","GuestOSFullName","CpuCount","MemoryGB")
            foreach ($h in $headers) {
                $ps.Append("th",$null,$h).N()
            }
            $ps.Up().N()

            [Array] $VMs = $Vapp | Get-CIVM

            foreach ($VM in $VMs) {
                $ps.Add("tr").N()
                $ps.Append("td",$null,$VM.Name).N()
                $ps.Append("td",$null,$VM.Status).N()
                $ps.Append("td",$null,$VM.GuestOSFullName).N()
                $ps.Append("td",$null,$VM.CpuCount).N()
                $ps.Append("td",$null,$VM.MemoryGB).N()

                $ps.Up().N()

            }
            $ps.Up().Up().N()

        }
        $ps.Up().N()

    }
    $ps.save($Path)

    "$(Get-Date -Format "yyyy-MM-dd HH:mm:ss") - Open HTML Report..."
    Start-Process $Path

}
}
```

The complete Module is also available as a GitHub repository.

GET THE MODULE

I love automation, so I created a simple script to get the latest release of the vCloud Director Tenant HTML Report Module:

```powershell
# Download
wget https://github.com/mycloudrevolution/VMware-vCD-TenantReport/archive/master.zip -OutFile C:\temp\master.zip

# Unzip
Add-Type -AssemblyName System.IO.Compression.FileSystem
function Unzip {
    param([string]$zipfile, [string]$outpath)
    [System.IO.Compression.ZipFile]::ExtractToDirectory($zipfile, $outpath)
}

Unzip C:\temp\master.zip C:\temp\

# Import
Import-Module "C:\temp\VMware-vCD-TenantReport-master\VMware-vCD-TenantReport.psd1"
```

#PSBlogWeek Wrap-Up

This week something special happened in the PowerShell community. It brought forth a third round of the popular blogging event known as #PSBlogWeek. Born from Jeff Hicks' idea nearly a year ago, #PSBlogWeek was started to get more people sharing their knowledge with the PowerShell community through blogging. Six willing volunteers to took their valuable time and posted a technical, content-rich article on their blog all coordinated around a central theme.

#PSBlogWeek is special because it's simply not about one person and a blog. It's about a coordinated effort between multiple people not only to write the articles but also to promote them as well. #PSBlogWeek is a way to start giving back that knowledge you have stored in your noggin and is a great way to get some exposure to yourself and your writing. #PSBlogWeek is heavily promoted on Twitter by many of the top PowerShell personalities in the community as well as by the participants themselves. If you're looking for an excuse to start blogging on PowerShell and want to be sure people see it, #PSBlogWeek is a great opportunity.

This week's theme was Server Management. Big thanks to the following participants for their contributions. Be sure to check out all of the great posts if you haven't already and let the authors know via Twitter if they helped you learn a thing or two.

Monday (Dan Franciscus @dan_franciscus) – Automating Chocolatey Package Internalizing With PowerShell
(Darwin Sanoy @DarwinTheorizes) – Logging and Error Handling Best Practices for Automating Windows Update Installs

Tuesday (Joshua King @windosnz) – Creating Storage Reports With Powershell

Wednesday (Josh Duffney @joshduffney) – Using Desired State Configuration (DSC) Composite Resources

Thursday (Volker Bachmann @VolkerBachmann) – Migration of SQL Server With PowerShell dbatools

Friday (Markus Kraus @vMarkus_K) – Using PowerShell to Create a vCloud Director Tenant HTML Report

Editing by: Sarah Cisco

#PSBlogWeek events are not held on a particular schedule but if you're a blogger and want to be part of the next one, contact Adam Bertram (@adbertram) on Twitter. He will be gathering a list of interested participants to potentially be notified the next time the event is held.