SOGETI

Introductie Powershell

Juli 2016

Introduction Powershell

Variabelen

Een variabelen is simpel gezegd een doos met een naam in het geheugen van de computer.
Je kunt van alles in deze doos stoppen. Een computernaam, een verzameling processen, een XML document etc.
Je kunt de doos openen door middel van de naam. Als je de doos geopend hebt kun je er dingen in stoppen of uithalen.
Let op: wat in de doos zit blijft in de doos.
We kunnen variabelen gebruiken voor

- Opslaan van informatie die je later wilt gebruiken
- Opslaan resultaat uit een commando of een draaiend script

Opslaan van waardes in een variabelen
Zo goed als alles in Powershell wordt behandeld als een object. Zelfs een simpele string of een karakter (bv. een computernaam) is een object.

```
PS C:\> "computername" | get-member
```

Het voorbeeld hierboven laat zien dat het object is van het type system.string en dat het veel eigenschappen (properties) en methodes heeft waar je mee kunt werken.

Ondanks dat technisch gezien een string een object is (zoals alles in de shell) zijn er mensen die het gewoon een simpele waarde noemen. Dat is omdat je je in de meeste gevallen alleen druk maakt om de string en niet om zijn properties (bv. bij computernaam). Bij een proces is dat anders omdat je daar rekening houdt met individuele eigenschappen zoals VM,,Name,,CPU, etc.

```
$hello = "hello"
$process = get-process
```

In Powershell is er weinig verschil tussen commandline en script

Voordat we verdergaan houd in gedachten dat variabelen in Powershell niet alleen in een script werken. In Powershell is er namelijk heel weinig verschil tussen command line en script.
Je kunt dus in de shell gewoon een variabelen definiëren en er een waarde aan hangen. Bv. $x = 1
De variabelen is dus X. In Powershell geef je namelijk aan dat wat na het $-teken komt een variabele is.

In de huidige vorm van werken is de GUI niet weg te denken maar Microsoft weet ook dat we dingen willen automatiseren.

Namen van Variabelen:

- bevatten normaal gezien letters, nummers en underscores. Het is gebruikelijk om te beginnen met een letter of een _
- kunnen spaties bevatten maar de naam moet dan tussen {} staan ${Mijn variabele}
- Variabelen bestaan niet tussen shell sessies. Dus als je de shell sluit zijn de variabelen verdwenen.
- Speciaal voor VBScripters, De Powershell community heeft er voornamelijk voor gekozen om niet het type mee te geven in de variabele naam. Geen str in een str object. bv $strComputers gebruik dus de $ gevolgd door de variabelennaam

Gebruik dus de $ gevolgd door de naam van de variabele.

Voorbeeld
get-wmiobject win32_computersystem -comp SERVER-R2
$var = "SERVER-R2" get-wmiobject win32_computersystem -comp $var

Variabelen

Als je strings gebruikt moet je deze tussen enkele of dubbele aanhalingsteken zetten. Er is een klein verschil.

Voorbeeld
$x = "hello"
$x = 'Hello'

Het verschil tussen enkele en dubbele aanhalingstekens is dat PowerShell alles tussen enkele aanhalingstekens behandelt als een letterlijke tekenreeks.

bekijk nu het volgende voorbeeld:

```
PS C:\> $var = 'What does $var contain?'
PS C:\> $var
What does $var contain?
PS C:\>
```

Powershell, een nieuwe manier van beheren.

Je ziet dat $var binnen enkele aanhalingstekens wordt behandeld als letterlijk tekst, Maar bij dubbele aanhalingstekens is dit niet het geval.

Bekijk het volgende voorbeeld.

```
PS C:\> $computernaam = 'SERVER-R2'
PS C:\> $zin = "De computernaam is $computernaam"
PS C:\> $zin
De computernaam is SERVER-R2
PS C:\>
```

Extra Voorbeeld
$one = "World"
$a = "hello, $one"
$b = 'hello, $one'

Opslaan van meerdere objecten

Tot dit punt hebben we gewerkt met variabelen die 1 object bevatten, die allemaal een eenvoudige waarde hebben.
We hebben direct gewerkt met het object zelf, in plaats van met de eigenschappen en/of methoden.
Laten we eens proberen meerdere objecten in een variabele stoppen.
Een manier om dit te doen is het gebruik van een door komma's gescheiden lijst.
PowerShell herkent deze als een verzameling (lijst)van objecten.

```
PS C:\> $computers = 'SERVER-R2', 'SERVER1', 'localhost'
PS C:\> $computers
SERVER-R2
SERVER1
localhost
```

Let op dat de komma's buiten de aanhalingstekens staan.
Eigenlijk is dit een array. Je verwijst dus naar je array index tussen de haakjes

```
PS C:\> $computers[0]
SERVER-R2
PS C:\> $computers[1]
SERVER1
PS C:\> $computers[-1]
localhost
PS C:\> $computers[-2]
SERVER1
PS C:\>
```

Denk bij een array aan een trein met verschillende rijtuigen en in elk een rijtuig een lading.

.

De variabelen zelf hebben een eigenschap die kan laten zien hoeveel objecten er zijn : (zie $computers | GM)

```
PS C:\> $computers.count
3
```

Buiten deze speciale eigenschap, hebt je ook toegang tot de eigenschappen en methoden van de objecten in de variabele alsof de eigenschappen en methoden van de variabele zelf zijn. Dit is makkelijker te begrijpen als je het ziet.
($computername = “Server-R2”)

```
PS C:\> $computername.length
9
PS C:\> $computername.toupper()
SERVER-R2
PS C:\> $computername.tolower()
server-r2
PS C:\> $computername.replace('R2','2008')
Server-2008
```

Opslaan van meerdere objecten

Het escape karakter oftewel *backtick*, zorgt ervoor dat het volgende karakter na dit escape karakter als letterlijke karakter wordt gezien.

Voorbeeld	
cd \program files	Werkt niet want hij ziet het als 2 stukken
cd '\program file'	Werkt
cd \program` files	Werkt ook . De backtick verteld powershell dat er niet 2 losse parameters aan het cmdlet set-location (cd) worden meegeven
$value = 123	
$zin = "De variabele $value bevat $value"	
$zin = "De Variabele '$value bevat $value"	

Een min put van de *backtick* is dat deze moeilijk te lezen is in de commandline. Een ander of groter font kan dit duidelijker maken.

Variabalen operators

• -as	100 / 3 -as [int]
• -is	"name" -is [string]
• -replace	"hostname2" –replace "2", "3"
• -join	1,2,3 -join " | "
• -split	"1,2,3" -split ","
• -in	3 -in 1,2,3
• -contains	1,2,3 -contains 3

De "command-line" Interface versus ISE (Interactive Scripting Environment

Object collections in de pipeline

We gaan het hebben over twee verschillende manieren waarop je kunt werken met objecten.
We gaan kijken naar wat object collecties precies zijn. Daarna gaan we kijken hoe we kunnen werken met object groepen tegelijk en met objecten in groepen individueel
Tenslotte gaan we het For-each cmdlet bekijken in samenwerking met WMI .

Object Collecties

Wat is een Object Collectie? Geloof het of niet maar waarschijnlijk ben je het al een aantal keer tegen gekomen.
Wanneer je bijvoorbeeld het cmdlet Get-service draait dan is dat een collectie van objecten.
Een collectie is eigenlijk meer een hip word voor, laat eens zeggen een groep van iets. Dus als we meer dan één service object hebben dan is die hele groep bij elkaar een collectie.
Powershell kan met deze objecten op verschillende manieren werken..
Het kan een actie naar een hele groep verzenden of naar een individueel object in de collectie. We hebben al een paar voorbeelden gezien met hoe je een actie naar een hele groep van objecten kunt uitvoeren.

Get-service genereert een groep met service objecten en "piped" de output naar stop-service die iedere individuele service stopt.

laten we eens kijken naar een paar voorbeelden.

Voorbeelden
PS c:\>Get-service
PS c:\>Get-service | Where-object { $_.status –eq "stopped"}
PS c:\>Get-service | Where { $_.status –eq "stopped"} | Start-service –whatif
PS c:\>Get-service | Where { $_.name -eq "winRM"} | Start-service –whatif

Get-service produceert een collectie van objecten. Na het *Where-object* filter heb je dus nog steeds een collectie alleen kleiner.
Start service is dus instaat om de hele collectie uit
Where-object { $_.status –eq "stopped"} te halen en probeert daarna om iedere service te starten. Één voor één probeert hij de services te herstarten. Dit kun je zien in d.m.v. de -Whatif output. (op alfabetische volgorde)
Een ander voorbeeld

Voorbeelden
PS c:\>Dir *.mp3 –recurse | Del

Hierbij zoeken we alle MP3 files op de computer en gaat deze verwijderen. Dit is een collectie met file system objecten.

Objecten individueel
Soms willen we ook werken met individuele objecten in groepen. Stel je voor dat we bijvoorbeeld een hele lijst met computers willen herstarten. We weten dat WMI ons die mogelijkheid bied, maar dit moeten we wel in stapjes doen. (WMI behandelen we in Avond 4)
In de folder c:\demo maken we de tekstfile met de naam "Computers.txt". Daarin zetten we een aantal computernamen(onder elkaar).

PS c:\Demo>Type computers.txt

Het werkt als volgt. (Typ is eigenlijk Get-content. Type is een Allias.)
Type (get-content) produceert meerdere objecten. Een collectie van objecten. Elk object wordt gezien als een string object . Deze namen noem ik even Computername1, Computername2 en Computername3. Voor elke Computername willen we een connectie maken naar WMI. We willen iets opvragen uit de class win32_operatingsystem maar hiervoor moeten we naar iedere computer individueel een verbinding opzetten. Dit kan niet simultaan of parallel.
Dit genereert een WMI Object. Zeg maar WMI-A, WMI-B en WMI-C.
Voor ieder WMI object dat we terugkrijgen moeten we een methode starten van dat object dat uiteindelijk de computer kan herstarten.

Get-Content		
	WMI A	Reboot
Computername1	WMI B	Restart
	WMI C	Logoff
	WMI D	
Computername2	WMI E	
	WMI F	
	WMI G	
Computername3	WMI H	
	WMI I	

Welke WMI objecten er ook geproduceerd worden je zult hier ook wat mee moeten doen (Dit is nesting)

We zijn dus in staat om meerdere WMi-objecten terug te krijgen van iedere Computer. Dit creëert een soort hiërarchie.
Get-Content (Alias GC) produceert 1,2,3 objecten. Computernaam objecten. Naar deze objecten willen we een WMI connectie maken naar de win32_operatingsystem class. Voor het object dat je terug krijgt wil je de restart method starten.

→ laat de methodes en properties zien.

```
GC computers.txt | For-each-object  {
        GWMI win32_operatingsystem  -comp $_  |
                For-each-object { $_.reboot() }
}
```

Voor elk WMI object dat we terug krijgen start je een method die er voor zorgt dat de computer wordt herstart.
Eerst willen we iets doen met elke naam die get-content produceert, Welke WMI objecten er ook geproduceerd worden je zult hier ook wat mee moeten doen.
Het commando dat we hier voor gebruiken is ForEach-Object

```
Get-content <Filename > |
   ForEach-object
      { GMWI <Class>  -comp $_  }
```

Achter Get-content specificeren de filename waar de computernamen in staan die we willen herstarten.
Deze pipen we vervolgens naar ForEach-Object zodat we met elk van hen iets kunnen doen.
Vervolgens specificeren we een script block dat ons verteld wat we met iedere computernaam gaan doen. We willen connecten naar WMI. Daarvoor gebruiken de Get-WMIObject cmdlet , een specifieke class en we willen dit voor een remote computer dus gebruiken we de –computer parameter.
De computer die we willen connecten gaan één voor één in $_ .
$_ vertegenwoordigd het huidige pipeline object.
Dus als get-content uit deze computers.txt 3 computernamen produceert, dan is dit wat de pipeline in gaat. Foreach-object krijgt , 3 Objecten en dus zal het scriptblock 3x gestart worden.

Dus als get-content uit deze file 3 computernamen produceert en dat is wat de pipeline in gaat

GC computers.txt | For-each-object { GMWI win32_operatingsyste -comp $_ }

We moeten nu iets gaan doen met de WMI objecten. Voor ieder WMI object dat we terugkrijgen, pipen we de output vervolgens naar een andere instance van Foreach-object. Dit is nesting. Dus we voegen een pipe en een nieuwe Foreach-object toe met een scriptblock. Dus tussen {} en we gaan $_ welke op dit moment een WMI object voorstelt en niet een computernaam.
Daar komen we zo op terug. De $_ willen we herstarten dus dat wordt $_.reboot()

Dit is wat er gebeurd, Get content draait eerst en produceert 3 objecten die de pipeline in gaan. De 3 objecten gaan naar ForEach-object. Dit scriptblock draait dus 3 maal $_ en word dus drie maal voorzien van een computernaam. Het cmdlet Get-WMIObject produceert WMI objecten. De WMI objecten worden vervolgens weer in de pipeline doorgestuurd naar de volgende instance van ForEach-object. Deze ForEach-object cmdlet krijgt WMI objecten en deze gaan in de $_ en hebben een reboot Methode wat gestart word.

```
GC computers.txt | For-each-object  {
                        GMWI win32_operatingsystem  -comp $_  |
                                For-each-object { $_.reboot() } }
```

Alias Foreach object = %

Extra Voorbeeld, resetten van password op een service

```
GC computers.txt | % {
        GWMI win32_service –comp $_ -filter “name='MSSQLSERVER'  |
         % {$_.change (,,,,,,,"P@ssword")}
}
```

www.google.nl → Win32_operatingsystem → reboot

Scripting

Powershell is een commandline interface en scripttaal. Er is eigenlijk geen verschil tussen deze twee. Wat je in de commandline kunt kun je in een script en vis versa. Een script, net zoals bij een acteur, verteld ons wat er moet gebeuren.
Voor dat je begint met scripten zijn er een aantal dingen die je moet weten.

Maak gebruik van een goede editor (en niet Notepad!!). Welke je wilt gebruiken is aan jezelf. Het gebruik van een tekst editor maakt het debuggen gemakkelijk. Hieronder een paar voorbeelden. Er zijn er nog meer te vinden op internet.

- Powershell ISE
- PowerGUI (Quest)
- Notepad ++
- PrimalForms (Niet Gratis)

Maak gebruik van een texteditor (en niet Notepad!!).

Standaard wordt het niet toegelaten om scripts zomaar te draaien.
Voor de duidelijkheid, Powershell voegt geen lagen toe als het om security gaat. Als je het qua rechten niet in de GUI kan, kan je het ook niet met powershell.
De powershell beveiliging is er voornamelijk om te voorkomen dat iemand onbedoeld een script draait. Zoals met het vbscripts vaak genoeg gebeurde.
Er zijn 3 manieren die dit voorkomen.

1. Als je dubbelklikt op een .ps1 file dat gaat deze naar notepad en niet naar powershell. Dit moet je ook gewoon zo laten staan.
2. Een scripts in powershell starten vanuit een directory met alleen de naam en zonder het pad werkt niet.
   Er zijn 2 manieren voor het volledige pad. Gebruik [tabcompletion] of zet het pad of .\ voor het powershell script.
3. Draai een script.ps1 en krijg de execution-policy error melding

Het draaien van scripts word standaard geblokkeerd. De algemene systeem setting, die voor iedereen op de computer werkt, kan alleen aangepast worden door iemand met administratieve rechten op het systeem. De rechten kunnen ook voor jezelf gezet worden. Dit wordt gedaan met de cmdlet Set-ExecutionPolicy.
Er zijn 5 niveaus en standaard staat deze op Restricted.

Policy	
Restricted	Alleen een standaard set van door Microsoft geleverde scripts die gebruikt worden om de PowerShell configuratie settings te zetten. Deze zijn gesigned door Microsoft.
All signed	laat alleen scripts toe die gesigned zijn en vertrouwd worden door een trusted CA.
Remotesigned	betekend dat alle lokale scripts gedraaid kunnen worden en alleen gesignde via het netwerk. Een UNC pad.
Unrestricted	betekend allles mag gedraaid worden, dit is niet aan te raden.
Bypass	is een speciale settings bedoeld voor ontwikkelaars die powershell in hun applicatie embedden. De execution policy wordt omzeilt hiermee.

Met de –scope parameter van de cmdlet kan je met bijvoorbeeld '-Scope personal' je eigen default aanpassen.

Scripts kun je zelf ook signen. Dan heb je een certificaat nodig dat vertrouwd wordt door je systeem. Precieze uitwerking is even out of scope van deze

Scripten:

We willen ons werk natuurlijk makkelijker maken. en tot nu toe hebben we alleen nog maar commando's ingetypt in de shell. Super! Jullie hebben al gezien hoeveel er met powershell kan zonder dat er een script is gemaakt.
Je moet scripten niet als programmeren zien maar als een manier om het makkelijker te maken om herhaaldelijk dezelfde commando's uit te kunnen voeren. Net als een batch script kan het simpel weg een opstapeling van een aantal commando's zijn.
Een script is simpelweg het kopiëren van een commando uit powershell in een tekst file.

Dus als get-content uit deze file 3 computernamen produceert en dat is wat de pipeline in gaat

We gaan nu werken vanuit ISE. Deze wordt default geleverd bij powershell
Let er op dat we de volledige cmdlets hebben gebruikt en elke parameter naam hebben gespecificeerd i.p.v. de positionele paramaters. Dit om het duidelijk te houden wat we precies doen en zodat we het later kunnen terug lezen en begrijpen of als iemand anders ons script gaat gebruiken.

We kunnen dit script draaien met de groene knop of F5 en kunnen ook losse regels selecteren en deze met F8 los draaien. Omdat dit commando netjes opgesplitst is kunnen we elke keer een regel extra selecteren en de tussen resultaten bekijken.
Dit commando kunnen we saven in bijvoorbeeld: Get-DiskInventory.ps1 en we hebben een script. Wel netjes met een cmdlet-stijl naam (werkwoord/zelfstandig naamwoord)
Wat nou als we dit zelfde commando willen hergebruiken maar dan voor een remote machine i.p.v. de gedefinieerde 'localhost'? Dit omdat een collega het ook wil gebruiken maar het toch wel een ingewikkeld commando voor hem/haar is.
Aangepaste versie naar ISE:

```
1   $computername = 'localhost'
2   |
3   Get-WmiObject -class Win32_LogicalDisk -computername $computername -filter "drivetype=3" |
4     Sort-Object -property DeviceID |
5      Format-Table -property DeviceID,
6           @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
7           @{label='Size(GB';expression={$_.Size / 1GB -as [int]}},
8           @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
9
```

We gaan nu het script parametriseren zodat het script gestart kan worden met een paramater

```
1   Param ($computername = 'localhost')
2   |
3
4   Get-WmiObject -class Win32_LogicalDisk -computername $computername -filter "drivetype=3" |
5     Sort-Object -property DeviceID |
6      Format-Table -property DeviceID,
7           @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
8           @{label='Size(GB';expression={$_.Size / 1GB -as [int]}},
9           @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
10
```

Je kunt het script nu starten via onderstaande manier

```
PS C:\demo> .\Get-DiskInventory.PS1 -computername localhost
```

Documenteren kan op meerdere manieren. Maar je moet het wel doen. Maakt het duidelijk voor zowel jezelf als voor een andere. Standaard manier van commentaar regels is altijd goed.
Maar powershell heeft ook nog een toegevoegde manier van Comment Syntax. Dit zorgt ervoor dat het help commando gebruikt kan worden om help te krijgen over je geschreven script.

Voege onderstaande toe aan het script

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more computers.
It displays each disk's drive letter, free space, total size, and percentage of free space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>


 Param ($computername = 'localhost')


 Get-WmiObject -class Win32_LogicalDisk -computername $computername -filter "drivetype=3" |
   Sort-Object -property DeviceID |
    Format-Table -property DeviceID,
        @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
        @{label='Size(GB';expression={$_.Size / 1GB -as [int]}},
        @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

Vraga de help op van dit script

```
PS C:\demo> get-help .\Get-DiskInventory.PS1
```

Onderstaande help zal verschijnen.

```
PS C:\demo> get-help .\Get-DiskInventory.PS1

NAME
    C:\demo\Get-DiskInventory.PS1

SYNOPSIS
    Get-DiskInventory retrieves logical disk information from one or more computers.


SYNTAX
    C:\demo\Get-DiskInventory.PS1 [[-computername] <Object>] [<CommonParameters>]


DESCRIPTION
    Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more compute
    free space, total size, and percentage of free space.


RELATED LINKS

REMARKS
    To see the examples, type: "get-help C:\demo\Get-DiskInventory.PS1 -examples".
    For more information, type: "get-help C:\demo\Get-DiskInventory.PS1 -detailed".
    For technical information, type: "get-help C:\demo\Get-DiskInventory.PS1 -full".
```

We kunnen nog een aantal verbeterslagen toevoegen aan een script.

- Geavanceerde opties [CmdletBinding()]
- Verplichte parameters [Parameter(Mandatory=$True)]
- Parameter aliasen [Alias('hostname')]
- Parameter validatie [ValidateSet(2,3)]
- Verbose output Write-Verbose (en -verbose)

We gaan ze nu een voor een toevoegen

Geavanceerde opties

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more compu
It displays each disk's drive letter, free space, total size, and percentage of free space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>

[CmdletBinding()]


 Param ($computername = 'localhost')


 Get-WmiObject -class Win32_LogicalDisk -computername $computername -filter "drivetype=3" |
   Sort-Object -property DeviceID |
    Format-Table -property DeviceID,
```

Een parameter Mandatory (verplicht) maken

```
param ([Parameter(Mandatory=$True)]
        $computername
    )
```

Als je nu het script runt zonder parameter toevoeging krijg je onderstaande

```
PS C:\demo> .\Get-DiskInventory.PS1
cmdlet Get-DiskInventory.PS1 at command pipeline position 1
Supply values for the following parameters:
computername:
```

Dus als get-content uit deze file 3 computernamen produceert en dat is wat de pipeline in gaat

help message toevoegen aan mandatory parameter

```
param ([Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")]
        $computername
    )
```

Als je nu het script draait zonder parameter toevoeging krijg je onderstaande

```
PS C:\demo> .\Get-DiskInventory.PS1
cmdlet Get-DiskInventory.PS1 at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
computername:
```

Als je nu !? invult krijg je de help message te zien

```
PS C:\demo> .\Get-DiskInventory.PS1
cmdlet Get-DiskInventory.PS1 at command pipeline position 1
Supply values for the following parameters:
(Type !? for Help.)
computername: !?
Enter a computer name to query
computername:
```

Alias toevoegen aan parameter

```
param ([Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")]
        [Alias('hostname')]
        $computername
    )
```

```
PS C:\demo> .\Get-DiskInventory.PS1 -computername localhost

DeviceID                                                        FreeSpace(MB)
--------                                                        -------------
C:                                                                      30347
M:                                                                        598

alias

PS C:\demo> .\Get-DiskInventory.PS1 -hostname localhost

DeviceID                                                        FreeSpace(MB)
--------                                                        -------------
C:                                                                      30347
M:                                                                        598
```

Validatie set toevoegen

```
param ([Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")]
        [Alias('hostname')]
        [string]$computername,

        [ValidateSet(2,3)]
        [int]$drivetype = 3
    )
```

Als je nu het script starten je wil drivetype 5 weten dan krijg je een error omdat je driveletter 2 en 3 als validatie set hebt staan. 5 is geen 2 of 3!

```
PS C:\demo> .\Get-DiskInventory.PS1 -hostname localhost -drivetype 5
C:\demo\Get-DiskInventory.PS1 : Cannot validate argument on parameter 'drivetype'. The argument "5" does not belong to the set "2,3"
ValidateSet attribute. Supply an argument that is in the set and then try the command again.
At line:1 char:56
+ .\Get-DiskInventory.PS1 -hostname localhost -drivetype 5
+                                                        ~
    + CategoryInfo          : InvalidData: (:) [Get-DiskInventory.PS1], ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,Get-DiskInventory.PS1
```

Verbose meldingen toevoegen

```
param ([Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")]
        [Alias('hostname')]
        [string]$computername,

        [ValidateSet(2,3)]
        [int]$drivetype = 3
    )

Write-Verbose "Connecting to $computername"
Write-Verbose "Looking for drive type $drivetype"

Get-WmiObject -class Win32_LogicalDisk -computername $computername `
 -filter "drivetype=$drivetype" |
 Sort-Object -property DeviceID |
 Select-Object -property DeviceID,
   @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
   @{name='Size(GB';expression={$_.Size / 1GB -as [int]}},
   @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}

Write-Verbose "Finished running command"
```

Als je het script draait met de – verbose parameter krijg je de uitgebreide meldingen te zien

Als je nu het script draait met de parameter –vebose krijg je de verbose meldingen te zien

```
PS C:\demo> .\Get-DiskInventory.PS1 -computername localhost -verbose
VERBOSE: Connecting to localhost
VERBOSE: Looking for drive type 3

DeviceID                                                  FreeSpace(MB)
--------                                                  -------------
C:                                                            30347
M:                                                              598
VERBOSE: Finished running command
```

We hebben nu ons eerste script gebouwd. Dit script is te gebruiken als een soort cmdlet. Dit door de toevoeging van de Help Syntax, de parameters etc.

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk instances from one or more computers.
It displays each disk's drive letter, free space, total size, and percentage of free space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3 -verbose
#>

 [CmdletBinding()]


 param ([Parameter(Mandatory=$True,HelpMessage="Enter a computer name to query")]
        [Alias('hostname')]
        [string]$computername,

        [ValidateSet(2,3)]
        [int]$drivetype = 3
    )


  Write-Verbose "Connecting to $computername"
  Write-Verbose "Looking for drive type $drivetype"


  Get-WmiObject -class Win32_LogicalDisk -computername $computername `
   -filter "drivetype=$drivetype" |
   Sort-Object -property DeviceID |
   Select-Object -property DeviceID,
     @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
     @{name='Size(GB';expression={$_.Size / 1GB -as [int]}},
     @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}

  Write-Verbose "Finished running command"

```