Netz-Weise
Lernen von den Besten.

Windows Powershell

Einführung + AD-Verwaltung

Version 1.0

Freundallee 13 a
30173 Hannover
www.netz-weise.de

Inhalt

Powershell Grundlagen

Windows Powershell wurde von Microsoft entwickelt, um sowohl die klassische Kommandozeile als auch VB-Skript als Skriptsprache zur Automatisierung abzulösen. Ab Windows 7 bzw. Windows Server 2008 R2 ist Powershell ins Betriebssystem direkt integriert. Seitdem bringt Microsoft mit jeder neuen Windows-Version auch eine neue Version der Powershell.

Powershell Version	Mitgeliefert ab	lauffähig ab
1.0	-	Windows XP Windows Server 2003 SP2
2.0	Windows 7 Windows Server 2008 R2	Windows XP Windows Server 2003 SP2
3.0	Windows 8.0 Windows Server 2012	Windows 7 SP1 Windows Server 2008 R2 SP1
4.0	Windows 8.1 Windows Server 2012 R2	Windows 7 SP1 Windows Server 2008 R2 SP1
5.0	Windows 10	Windows 7 SP1 Windows Server 2008 R2 SP1

Die Powershell kann jederzeit aktualisiert werden, indem man das Microsoft Management Framework installiert. Ab Version 2.0 bringt das Management-Framework auch Windows Remote Management mit. WinRM, wie das Remotemanagement auch abgekürzt wird, bringt die Möglichkeit mit, Windows Systeme auch aus der Ferne per Powershell zu steuern.

Powershell ist grundsätzlich zu großen Teilen kompatibel zur Windows Kommandozeile (cmd), so dass alle gängigen Kommandozeilentools in Powershell funktionieren. Mit Hilfe von sogenannten Aliasen und Funktionen sind sogar einige Unix-Befehle in Powershell nachgebildet. So kann man z.B. auch ls oder man aufrufen.

Powershell verfügt aber auch über eigene Kommandos, die sogenannten Cmdlets, Cmdlets gesprochen. Beim Design von Cmdlets wurde darauf geachtet, die Bedienung so konsistent und einfach wie möglich zu machen. Dafür wurden eine Reihe von Regeln aufgestellt:

Ein Cmdlet besteht immer aus einem Verb, das bestimmt, was das Cmdlet tut, und einem Objekt, das verarbeitet werden soll. Dadurch haben Cmdlets immer sprechende Namen. Das Verb und das Objekt sind durch ein Bindestrich direkt miteinander verbunden. Ein paar Beispiele für Verben sind:

- Get
- Set
- Add
- New
- Remove
- Clear
- Copy
- …

Alle verfügbaren Cmdlets lassen sich auch mit einem Cmdlet anzeigen, nämlich mit *Get-Verb*:

```
Get-Verb

Verb            Group
----            -----
Add             Common
```

```
Clear           Common
Close           Common
Copy            Common
Enter           Common
Exit            Common
Find            Common
Format          Common
…
```

Um sich alle verfügbaren Cmdlets anzeigen zu lassen, verwendet man ein anderes Cmdlet, *Get-Command*.

```
Get-Command

CommandType     Name
-----------     ----
Alias           Add-PefMessageProvider
Alias           Add-ProvisionedAppxPackage
Alias           Add-WAPackEnvironment
Alias           Apply-WindowsUnattend
Alias           Disable-WAPackWebsiteApplicationDiagnostic
Alias           Enable-WAPackWebsiteApplicationDiagnositc
Alias           Flush-Volume
Alias           Get-AzureStorageContainerAcl
Alias           Get-PhysicalDiskSNV
…
```

Im Beispiel sind nur die ersten Kommandos angezeigt, die tatsächlich gar keine Cmdlets sind, sondern Aliase. Ein Alias ist ein Verweis auf ein anderes Kommando. Gibt man das Alias ein, wird also stattdessen ein Cmdlet oder ein Programm gestartet. Alle Aliase zeigt – man mag es kaum glauben – das *Cmdlet Get-Alias* an:

```
Get-Alias

CommandType     Name
-----------     ----
Alias           % -> ForEach-Object
Alias           ? -> Where-Object
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           cat -> Get-Content
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
…
```

Auch hier wieder nur ein kleiner Ausschnitt, der aber schon zeigt, dass viele der alten cmd-Befehle Aliase sind, die auf ein Powershell-Cmdlet mit ähnlicher Funktionalität verweisen. *cd* ruft z.B. *Set-Location* auf. Aliase sollen nicht nur die Kompatibilität erhöhen, sondern auch das Tippen vereinfachen, wenn es sich um lange Kommandos handelt. Allerdings ist es nicht empfohlen, in Skripten Aliase zu verwenden, da man nie sicher sagen kann, ob ein Alias auf einem Fremdsystem nicht umgebogen wurde und sich anders verhält als auf einem System mit Powershell-Standard-Funktionalität.

Ein anderes wichtiges Design-Kriterium, das Microsoft in die Powershell hat einfließen lassen, betrifft Parameter. Parameter definieren Werte, die man dem Cmdlet übergeben kann, um deren Verhalten zu verändern. Dabei unterscheidet man zwischen einfachen Schaltern, die nur aktiviert oder deaktiviert werden können (wie bei einem Lichtschalter, der nur ein oder aus schaltet) und Parametern mit Übergabewerten. Die Übergabewerte werden korrekterweise als Argumente bezeichnet.
Parameter werden bei Powershell grundsätzlich mit einem Bindestrich eingefügt. Die Windows-Kommandozeile hat hier keine Vorgaben gemacht, so dass es 3 verschiedene Möglichkeiten gab, wie

ein Kommando Parameter übernommen hat: Mit dem Schrägstrich (Slash), mit einem Minus oder, eher selten, mit zwei Minus. Bei Powershell werden andere Parameterübergaben als der Bindestrich nicht verwendet. Es gibt nur eine Ausnahme: Bei sogenannten Positionsparameter können die Parameternamen komplett ausgelassen werden, und es wird nur das Argument übergeben.

```
Get-Command -Name get-h*

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-Help                                           Pscx
Cmdlet          Get-Hash                                           Pscx
Cmdlet          Get-Help                                           Microsoft...
Cmdlet          Get-History                                        Microsoft...
Cmdlet          Get-Host                                           Microsoft...
Cmdlet          Get-HotFix                                         Microsoft...
Cmdlet          Get-HttpResource                                   Pscx
```

Da der Parameter *-Name* ein Positionsparameter ist, kann der Name des Parameters, hier also –Name, ausgelassen werden:

```
Get-Command get-h*

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Function        Get-Help                                           Pscx
Cmdlet          Get-Hash                                           Pscx
Cmdlet          Get-Help                                           Microsoft...
Cmdlet          Get-History                                        Microsoft...
Cmdlet          Get-Host                                           Microsoft...
Cmdlet          Get-HotFix                                         Microsoft...
Cmdlet          Get-HttpResource                                   Pscx
```

Eine weitere Designvorgabe ist, dass ein Cmdlet mit möglichst wenig Parametern (möglichst keinem) eine Ausgabe erzeugt. Ein Beispiel ist *Get-Command*, der ohne Parameter alle verfügbaren Kommandos anzeigt. Gibt man den Parameter *-Name* ein, kann man die Ausgabe einschränken.
Zu guter Letzt sollte eine Cmdlet immer genau eine Funktion erfüllen, und nicht wie viele klassische Kommandozeilentools ein Füllhorn an Aufgaben, die dann aber nicht mehr überschaubar sind. Wenn der Name des Cmdlets gut gewählt ist, kann man dann direkt am Namen ersehen, was ein Cmdlet macht.

Vom Suchen und Finden von Cmdlets

Powershell bringt eine Unmenge von Cmdlets mit, die sich mit Hilfe von Modulen und Snap-Ins noch um ein vielfaches erweitern lassen. Das führt zu einer Riesen-Anzahl von Cmdlets, die sich wohl kaum jemand merken kann. Das wird zusätzlich erschwert durch die Tatsache, dass ein Cmdlet nicht mehr eine Unmenge von Aufgaben erfüllen soll, sondern spezialisiert ist. Nimmt man z.B. die dism.exe zum Bearbeiten von Windows-Images, so ist dieses als Powershell-Modul in Windows 8.1 auf 39 Cmdlets aufgeteilt worden, wobei die Cmdlets nicht einmal die volle Funktionalität von dism.exe erschlagen. Diese Menge an Cmdlets kann sich natürlich niemand mehr merken. Aber genau hier greift das Konzept der intelligenten Benennung, denn mit Hilfe von *get-command* kann man sich die Befehle, die man benötigt, meist schnell heraussuchen. Suchen Sie z.B. ein Cmdlet zum Aufrufen des Windows-Eventlogs, geben Sie einfach *get-command get-*event** ein

```
Get-command get-*event*

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Cmdlet          Get-Event                                          Microsoft...
```

```
Cmdlet          Get-EventLog                                        Microsoft...
Cmdlet          Get-EventSubscriber                                 Microsoft...
Cmdlet          Get-WinEvent                                        Microsoft...
[…]
```

Je nachdem, welche Windows-Komponenten und Module noch installiert sind, kann sich die Ausgabe unterscheiden. Wesentlich ist hier aber, dass Powershell Ihnen nur eine relativ geringe Anzahl von Ergebnissen zurück liefert. Um jetzt heraus zu finden, welches Cmdlet das richtige ist, gibt es ein weiteres wichtiges Cmdlet, *get-help*. *Get-Help* gibt Ihnen, wie der Name schon sagt, Hilfe, und zwar entweder über Powershell und allgemeine Powershell-Themen, oder über die einzelnen Cmdlets. Hierfür stellt *get-help* einen Positionsparameter *-Name* zur Verfügung, mit dem Sie angeben können, über welches Cmdlet oder welches Thema Sie Hilfe benötigen:

```
Get-Help –Name Get-Eventlog
```

Da es sich um einen Positionsparameter handelt, können Sie *-Name* aus auslassen.

```
Get-Help Get-Eventlog
```

Powershell gibt Ihnen jetzt eine rudimentäre Hilfe zum Befehl *Get-Eventlog* aus. Allerdings ist die Hilfe noch deutlich umfangreicher. Mehr Hilfe erhalten Sie über eine Reihe von Parameter, die *get-Help* mitbringt:

```
Get-Help Get-Eventlog –Full
Get-Help Get-Eventlog –Showwindow
Get-Help Get-Eventlog –Examples
Get-Help Get-Eventlog –Online
```

-Full zeigt die ausführliche Hilfe zum Befehl an, *-Showwindows* zeigt die Hilfe in einem Extra Hilfefenster an, dessen Darstellung sich auch anpassen lässt, und *-Examples* zeigt Ihnen Beispiele zur Verwendung des Kommandos. *-Online* ruft direkt die Hilfeseite des Cmdlets aus dem Internet ab – diese Option ist nützlich, wenn Sie die aktuelle Hilfe nicht heruntergeladen haben.

Updatefähige Hilfe

Das Herunterladen der Hilfe ist seit Powershell 3.0 notwendig. Bis Powershell 2.0 war die Hilfe integrierter Bestandteil der Powershell. Dann hat Microsoft die Updatefähige Hilfe eingeführt. Diese kann sich selbst aus dem Internet heraus aktualisieren und auch Fremdherstellermodule können mit einer neuen Hilfe ausgestattet werden. Um die Hilfe zum ersten Mal herunter zu laden oder zu aktualisieren, verwenden Sie den Befehl *Update-Help*:

```
Update-Help
```

Achten Sie darauf, dass die Hilfe nur mit administrativen Rechten gestartet werden kann, da die Hilfe-Dateien unterhalb des Windows-Ordners gespeichert werden, auf den normale Benutzer nur mit Lese-Rechten zugreifen können.

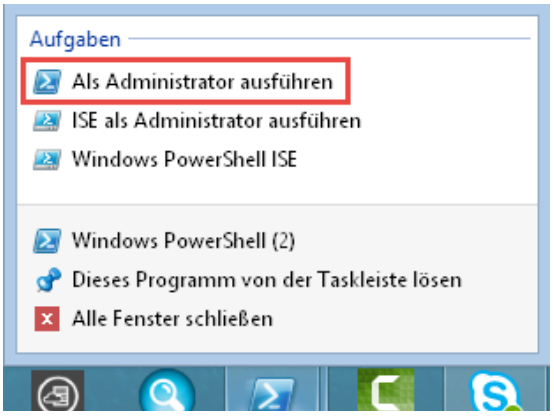


Abbildung 1- Powershell mit Admin-Rechten starten

Update-Help hat noch einen weiteren Nachteil – der Rechner, der aktualisiert werden soll, muss Internet-Zugriff haben. Hat er diesen nicht, schlägt die Aktualisierung fehl. Daher ist es möglich, die Hilfe auf einem PC herunter zu laden, der Internetzugriff hat, und die Hilfe dann über Update-Help mit dem Parameter *-sourcepath* zu starten:

```
Update-Help -sourcepath \\FS1\Daten\PSHHelp
```

Zum Herunterladen der Hilfe verwendet man das Cmdlet *Save-Help*, das auf einem PC mit Internetverbindung gestartet wird. *Save-Help* benötigt als Parameter den Pfad zu dem Ordner, in dem die Hilfe-Dateien gespeichert werden sollen.

```
save-help -DestinationPath \\fs1\Daten\PSHHelp
```

Save-Help durchsucht alle installierten Module auf dem Quellrechner und versucht dann, die Dateien in der Sprache des Rechners herunter zu laden, auf dem das Cmdlet ausgeführt wurde. Dies ist allerdings oft nicht die Hilfe, die man eigentlich benötigt – die Module der Server werden damit natürlich nicht berücksichtigt, und auch die Sprachversion ist auf einem Client eventuell eine andere. Dazu kommt, dass einige Module gar keine lokalisierte Hilfe besitzen. Zur Lösung des Sprachproblems kann man beim *Save-Help* einfach mit dem Parameter *-UICulture* die Sprache übergeben, die man herunterladen möchte:

```
save-help -DestinationPath \\fs1\daten\PSHHelp -UICulture en-us
```

Um die Module von Fremdrechnern herunter zu laden, muss man auf diesen erst die Modulinformationen exportieren und ein einer XML-Datei speichern. Dies funktioniert mit Hilfe der Befehle *Get-Module* und *Export-CliXML*. Die beiden Befehle werden mit Hilfe der Powershell Pipeline miteinander verknüpft. Die Funktion der Pipeline wird später beschrieben, hier ist erst einmal nur wichtig, dass die Daten von Get-Module direkt von Export-CliXML in eine Datei gespeichert werden, anstatt sie an der Kommandozeile auszugeben.

```
# Auf dem Quellserver auszuführen
get-module -ListAvailable | Export-Clixml -Path \\nwfs1\daten\modules.xml
```

Importieren Sie die Datei auf dem Rechner, auf dem *Save-Help* ausgeführt wird, und leiten Sie die Module per Pipeline weiter:

```
# Auf dem mit dem Internet verbundenen Rechner
Import-CliXML –Path \\nwfs1\daten\modules.xml | save-help –DestinationPath
\\fs1\daten\PSHHelp -UICulture en-us # Achtung, alles in einer Zeile eingeben!
```

Daten einlesen und ausgeben mit Powershell

Ein mächtiges Feature von Powershell ist der Daten Im-und Export. Im vorigen Beispiel haben Sie bereits gesehen, wie die Modul-Informationen in eine XML-Datei exportiert und wieder importiert worden sind. Powershell hat aber noch deutlich mehr Kommandos zum Im- und Exportieren von Kommandos.

Die wichtigsten Cmdlets sind:

```
Out-Gridview # Gibt Daten in einem Fenster aus
Export-CSV # Exportiert Daten als Trennzeichen-separierte Datei
Set-Content # Speichert Daten in einer Textdatei
Import-CSV # Importiert Daten aus einer Trennzeichen-separierten Datei
Get-Content # Liest Daten aus einer Textdatei aus.
```

Am einfachsten ist normalerweise die Arbeit mit der Powershell Pipeline – Die Pipeline arbeitet wie ein Fließband und leitet die Daten von einem Cmdlet direkt an ein anderes Cmdlet weiter. Für die Weiterleitung wird der senkrechte Strich verwendet:

```
get-eventlog -LogName Application -Newest 10 | Out-GridView
```

Der Befehl *Get-Eventlog* liest die Ereignisprotokolle aus, in diesem Fall das Anwendungs-Log. Außerdem werden nur die neuesten 10 Ereignisse ausgegeben. Die Ausgabe wird dann per Pipeline in das Cmdlet *Out-Gridview* weitergeleitet, dass die Daten in einem Fenster anzeigt:

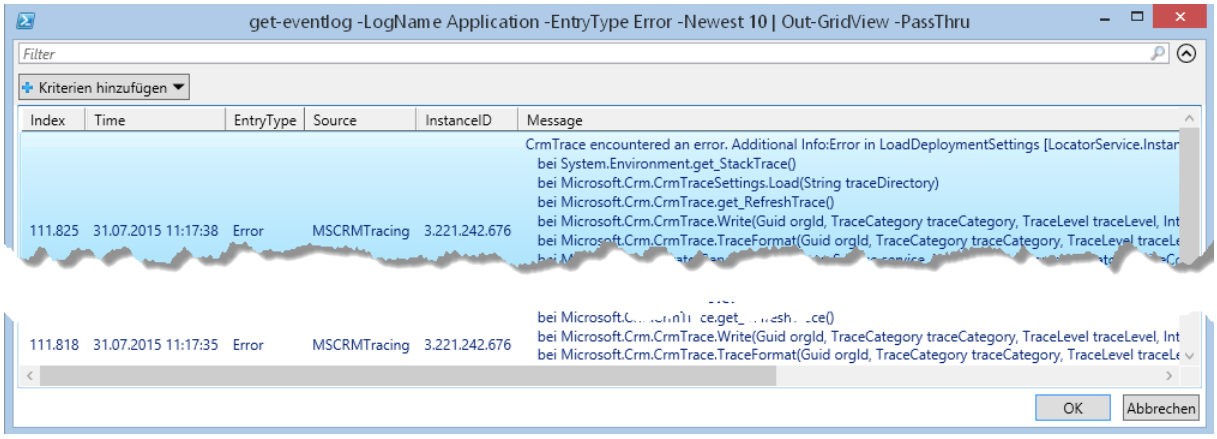


Abbildung 2 - Gridview zur Auswahl

Das Gridview stellt die Daten nicht nur übersichtlicher dar, sondern man kann mit einem Gridview die Daten auch weiter filtern, indem man entweder mit Hilfe des Feldes *Filter* über das gesamte Gridview sucht oder über *Kriterien hinzufügen* über einzelne Spalten filtert.

Das Gridview kann aber auch zur Auswahl einzelner Werte genutzt werden, indem man den Parameter *-Passthru* verwendet. Das Fenster bekommt nun noch einen zusätzlichen OK-Button.

```
get-eventlog -LogName Application -Newest 10 | Out-GridView -Passthru
```

Wählt man jetzt ein oder mehrere Datensätze aus dem Gridview mit der Maus aus und klickt OK, werden die Datensätze wieder in die Powershell zurückgegeben und können hier weiterverarbeitet werden. So kann man einfache grafische Oberflächen sehr schnell erzeugen.

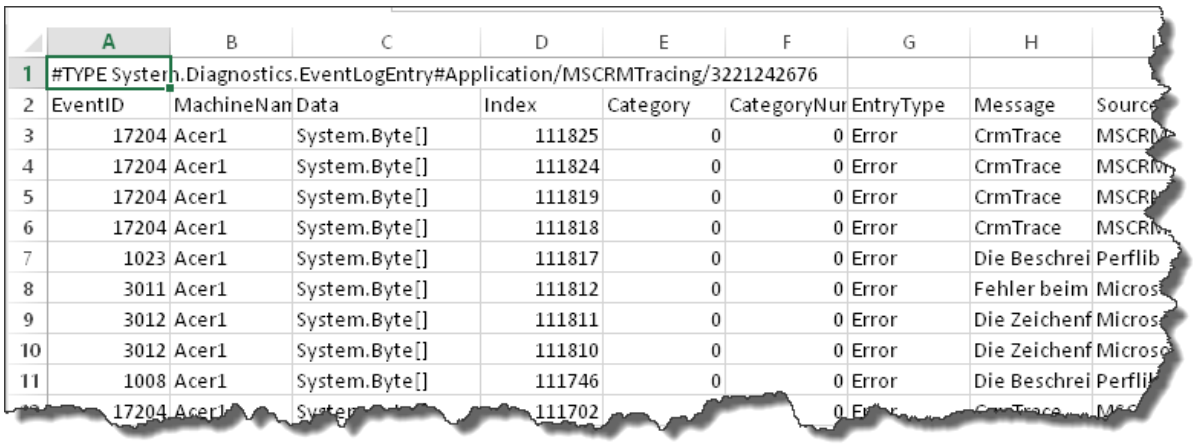
Will man die Daten weitergeben oder speichern, bietet sich Excel an. Hierfür stellt Powershell das Cmdlet *Export-Csv* zur Verfügung, da Excel in der Lage ist, CSV-Dateien direkt weiter zu verarbeiten. *Export-CSV* benötigt als Parameter mindestens den Pfad, in den die Ausgabedatei geschrieben werden soll:

```
get-eventlog -LogName Application -Newest 10 | Export-Csv –Path
C:\Export\Events.csv # die Zeile wurde automatisch umgebrochen!
```

Die CSV-Datei kann von Excel direkt verwendet werden. Allerdings nutzt Excel auf deutschen Systemen als Trennzeichen kein Komma, sondern ein Semikolon, so dass die Ausgabe von *Export-CSV* mit dem Parameter *-Delimiter* noch einmal angepasst werden muss:

```
get-eventlog -LogName Application -Newest 10 | Export-Csv –Path
C:\Export\Events.csv –Delimiter ';' # die Zeile wurde automatisch umgebrochen!
```

Nun kann die Datei direkt in Excel geöffnet werden.



	A	B	C	D	E	F	G	H	I
1	#TYPE System.Diagnostics.EventLogEntry#Application/MSCRMTracing/3221242676								
2	EventID	MachineNan	Data	Index	Category	CategoryNur	EntryType	Message	Source
3	17204	Acer1	System.Byte[]	111825	0	0	Error	CrmTrace	MSCRM
4	17204	Acer1	System.Byte[]	111824	0	0	Error	CrmTrace	MSCRM
5	17204	Acer1	System.Byte[]	111819	0	0	Error	CrmTrace	MSCRM
6	17204	Acer1	System.Byte[]	111818	0	0	Error	CrmTrace	MSCRM
7	1023	Acer1	System.Byte[]	111817	0	0	Error	Die Beschrei	Perflib
8	3011	Acer1	System.Byte[]	111812	0	0	Error	Fehler beim	Micros
9	3012	Acer1	System.Byte[]	111811	0	0	Error	Die Zeichenf	Micros
10	3012	Acer1	System.Byte[]	111810	0	0	Error	Die Zeichenf	Micros
11	1008	Acer1	System.Byte[]	111746	0	0	Error	Die Beschrei	Perfli
12	17204	Acer1	System.Byte[]	111702	0	0	Error	CrmTrace	MSCRM

Abbildung 3 - Die Ausgabe von Export-Csv kann in Excel

Import-CSV kann die Daten aus der CSV-Datei auch wieder importieren. Die Daten werden dann als Objekte direkt in Powershell importiert.

```
import-csv -Path C:\export\events.csv -Delimiter ';'

EventID             : 17204
MachineName         : Acer1
Data                : System.Byte[]
Index               : 111825
Category            : (0)
CategoryNumber      : 0
EntryType           : Error
Message             : CrmTrace encountered an error. Additional Info:Error in
                      LoadDeploymentSettings [LocatorService.Instance], Stack
                      Trace :    bei System.Environment.GetStackTrace(Exception
                      e, Boolean needFileInfo)
[…]
```

An dieser Stelle fallen Ihnen vielleicht zwei Dinge auf – zum einen gibt der *Export-Csv* mehr Informationen aus als der *Out-Gridview*. In der obigen Ausgabe finden Sie z.B. den Computernamen wieder, der im Gridview nicht angezeigt wird. Außerdem werden die Daten Eigenschaftenweise gespeichert. In Excel können Sie alle Informationen direkt Spaltenweise importieren. Das wäre z.B. mit einem einfachen Kommandozeilenbefehl gar nicht möglich. Nehmen Sie Beispielsweise die Ausgabe des Befehls ipconfig:

```
Ipconfig

Windows-IP-Konfiguration

Drahtlos-LAN-Adapter WiFi:
   Verbindungsspezifisches DNS-Suffix: fritz.box
   Verbindungslokale IPv6-Adresse  . : fe80::9160:5917:4892:5fd8%3
   IPv4-Adresse  . . . . . . . . . . : 192.168.178.41
   Subnetzmaske  . . . . . . . . . . : 255.255.255.0
   Standardgateway . . . . . . . . . : 192.168.178.1
```

Ipconfig liefert Ihnen Text zurück. Sie können auf die einzelnen Felder wie IPv4-Adresse oder Subnetzmaske nicht direkt zurückgreifen, ohne mit einem Kommandozeilentool wie *find* den Text zu durchsuchen. Würden Sie diese Daten in Excel importieren, würde die komplette Ausgabe in die erste Zelle geschrieben werden. Powershell erleichtert den Umgang mit Daten enorm, weil sie die einzelnen Daten auch separat speichert und weitergibt. Im Beispiel der Events wird z.B. für jedes Event ein eigener Datensatz angelegt und jede Information (Message, TimGenerated, MachineName,

EntryType) wird als separate Eigenschaft dieses Events gespeichert, ähnlich einer Excel-Tabelle. Man spricht dann von Objekten.

Wenn Sie tatsächlich nur Text haben, den Sie verarbeiten möchten, hilft Ihnen das Cmdlet *Set-Content* zum Speichern und *Get-Content* zum Einlesen weiter. Zum Einlesen der WindowsUpdate.log im Windows-Ordner nehmen Sie Get-Content:

```
get-content C:\windows\WindowsUpdate.log
```

Wenn Sie die Ausgabe des Ipconfig in eine Textdatei umleiten möchten, verwenden Sie Set-Content:

```
ipconfig /all | set-content c:\export\ip.txt
```

Interessanterweise ist die WindowsUpdate.Log eigentlich auch eine Komma-separierte Datei. Als Trennzeichen wird allerdings der Tabulator verwendet. Der Tabulator wird in Powershell durch ein t, angeführt vom Backtick (`) angegeben, also `t. Versuchen Sie doch mal, die WindowsUpdate.Log mit Import-Csv zu lesen und dann mit Hilfe des Gridviews auszugeben. Dazu benötigen Sie den Parameter –Header, um die Namen der einzelnen Spalten der csv-Datei anzugeben.[1]

Variablen und Objekte

Um Daten wiederzuverwenden, auf einzelne Eigenschaften von Daten zuzugreifen oder Daten in Skripten variabel zu verwenden, stellt Powershell Variablen zur Verfügung. Variablen haben einen Namen, über den immer wieder auf die Daten zugegriffen werden kann. Zum Erzeugen einer Variablen stellen Sie dem Namen der Variablen, den Sie bis auf wenige Voragen frei vergeben können, einfach ein $ voran. Um alle Variablen anzuzeigen, die im System bereits deklariert sind, verwenden Sie das Cmdlet *Get-Variable*.

```
Get-Variable

Name                           Value
----                           -----
$                              Get-Variable
?                              True
^                              Get-Variable
args                           {}
ConfirmPreference              High
ConsoleFileName
[…]
```

Variablen müssen in Powershell nicht vor der Benutzung erstellt werden, sondern Sie können in einem Zug deklariert und verwendet werden. Möchten Sie beispielsweise die Events aus dem obigen Beispiel zwischenspeichern, legen Sie den Namen der neuen Variablen fest, stellen ein Gleichheitszeichen hinter die Variable und schreiben dahinter den Befehl, dessen Ausgabe Sie festhalten möchten. Powershell speichert die Ausgabe des Befehls dann direkt in die Variable:

```
$Events = get-eventlog -LogName Application -Newest 10
```

Nun können Sie in der Powershell einfach *$Events* eingeben, und die Ausgabe wird zurückgegeben. Da die Variable die Events jetzt enthält, können Sie *$Events* beliebig oft aufrufen, Sie erhalten jedes Mal die gleiche Rückgabe. Der Befehl wird hierzu nicht wieder aufgerufen.

```
$Events

   Index Time          EntryType   Source                 InstanceID Message
   ----- ----          ---------   ------                 ---------- -------
  111902 Jul 31 13:13  Information ESENT                         103 svchost...
```

[1] Import-CSV –Path C:\Windows\WindowsUpdate.log –Delimiter "`t" –Header "Date","Time","PID",TID","Component","Text" | Out-Gridview

```
  111901 Jul 31 13:13  Information ESENT                          327 svchost...
  111900 Jul 31 13:08  Information ESENT                          326 svchost...
  111899 Jul 31 13:08  Information ESENT                          105 svchost...
  111898 Jul 31 13:08  Information ESENT                          102 svchost...
  111897 Jul 31 12:29  Information ESENT                          103 svchost...
  111896 Jul 31 12:29  Information ESENT                          327 svchost...
  111895 Jul 31 12:28  0           Software Protecti...   1073742727 Der Sof...
  111894 Jul 31 12:28  Information Software Protecti...   1073758208 Der Sof...
  111893 Jul 31 12:28  Information Software Protecti...   1073742827 Der Sof...
```

Powershell versucht selbständig herauszufinden, welche Daten in den Variablen gespeichert werden und legt daraufhin den Datentyp fest, der verwendet werden muss. Um herauszufinden, welchen Datentyp Powershell verwendet hat, geben Sie den Variablennamen an, gefolgt von einem *.gettype()*:

```
$Events.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Wir sehen hier, dass es sich um eine Variable vom Typ Array handelt. Ein Array ist eine Auflistung von Objekten. Das liegt daran, dass wir eine ganze Reihe von Events direkt in die Variable geleitet haben. Einzelwerte haben andere Datentypen wie z.B. Int (Ganzzahlen), Datetime (Datum und Uhrzeit) oder String (Text). Wenn Sie z.B. einen Text wie einen Pfad speichern wollen, müssen Sie den Text in einfache oder doppelte Anführungszeichen stellen:

```
$Text = "Holger Voges"
$Text.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Unter Name finden Sie hier String. Da Strings nicht nur einfach Text sind, sondern selber wieder als Objekte gespeichert werden, wird als BaseType *System.Object* angegeben. Was das bedeutet stellen Sie fest, wenn Sie z.B. $Text.Length eingeben, denn dann bekommen Sie statt des Textes die Anzahl der Zeichen zurück, die der String beinhaltet:

```
$text.Length
12
```

Auch Variablen haben also, abhängig von der Vorlage (Klasse), von der Sie abgeleitet wurden, Eigenschaften, die den Inhalt der Variablen beschreiben. Zu Objekten können aber nicht nur Eingenschaften (feste Werte) gehören, sondern auch Methoden. Methoden sind kleine Programme, die den Inhalt des Objekts bearbeiten können und die Ihnen die Mühe abnehmen, solche Basis-Funktionalitäten selber zu schreiben. Methoden unterscheiden sich beim Aufruf von Eigenschaften durch ein () am Ende des Methodennamens – Methoden sind selbst kleine Programme und können, genau wie ein Cmdlet, Parameter übergeben bekommen, die aber einfach in die Klammern hinter der Methode geschrieben werden. Die Klammern sind also nur zur Übergabe von Parametern gedacht, sind aber ein eindeutiges Unterscheidungsmerkmal, da die Klammern immer angegeben werden müssen. Eine Methode haben Sie auch schon kennen gelernt – *gettype()*. Ein paar Beispiele für Methoden lernen Sie gleich noch mit dem Variablentyp *Datetime* kennen, aber erst noch mal zurück zu den Strings.

Auf den ersten Blick sieht es so aus, als würde es keinen Unterschied machen, ob sie einfache oder doppelte Anführungszeichen für einen String verwenden. Nehmen Sie nämlich statt doppelter in der obigen Kommandozeile einfache Anführungszeichen, ist das Ergebnis identisch:

```
$Text = 'Holger Voges'
$Text.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     String                                   System.Object
```

Der Unterschied zeigt sich erst auf den zweiten Blick. Versuchen Sie folgende Eingabe:

```
$Ansprache = "Guten Tag $Text"
$Ansprache
Guten Tag Holger Voges
```

Nehmen Sie stattdessen einfache Anführungszeichen, offenbaren sich die Unterschiede:

```
$Ansprache = 'Guten Tag $Text'
$Ansprache
Guten Tag $Text
```

Mit doppelte Anführungszeichen werden alle Powershell-spezifischen Sonderzeichen ausgewertet. Verwenden Sie einfache Sonderzeichen, speichert Powershell genau das, was sie eingegeben haben. Im obigen Beispiel wird die Variable *$Text* mit doppelten Anführungszeichen also in ihren Klartext aufgelöst, mit einfachen Anführungszeichen wird genau der Name ausgegeben.

Wenn Sie einer Variablen selber einen Datentyp zuweisen möchten, anstatt Powershell diese Arbeit zu überlassen, können Sie den Datentyp der Variablen einfach in eckigen Klammern zuweisen:

```
[String]$Ansprache = 'Guten Tag, Herr Voges'
```

Daten und Uhrzeiten speichern

Ein ebenfalls sehr nützlicher Variablentyp sind Datetime-Werte. Sie speichern Datums-und Zeitinformationen. Um Beispielsweise das aktuelle Datum zu speichern, verwenden Sie das Cmdlet *Get-date*.

```
$datum = get-date
$datum
Montag, 3. August 2015 10:36:45
```

Get-Date hat das aktuelle Datum in der Variablen $datum abgelegt. Das Datentyp, den Powershell hierfür verwendet hat, ist datetime.

```
$datum.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DateTime                                 System.ValueType
```

So ein Datetime-Objekt stellt eine ganze Reihe von interessanten Eigenschaften zur Verfügung. Um die Variable $Date zu untersuchen, lassen Sie sich alle Eigenschaften von Datum auflisten. Das geht mit dem Cmdlet *Select-Object*, dessen Funktionswiese wir im Abschnitt über die Pipeline noch einmal genauer beleuchten.

```
$datum | Select-Object -Property *

DisplayHint : DateTime
DateTime    : Montag, 3. August 2015 10:38:19
Date        : 03.08.2015 00:00:00
Day         : 3
DayOfWeek   : Monday
DayOfYear   : 215
Hour        : 10
Kind        : Local
```

```
Millisecond : 240
Minute      : 38
Month       : 8
Second      : 19
Ticks       : 635741950992400310
TimeOfDay   : 10:38:19.2400310
Year        : 2015
```

Sie sehen, dass das Datums-Objekt nicht nur das Datum speichert, sondern auch die Uhrzeit. Da die einzelnen Komponenten des Datums in separaten Eigenschaften gespeichert sind, kann man die Einzelkomponenten auch direkt abrufen. Dazu geben Sie hinter dem Variablennamen einen Punkt an, und dann die Eigenschaft, auf die Sie zugreifen möchten. Der Punkt bedeutet immer, dass Sie nicht die Standardausgabe des Objekts haben möchten, sondern eine Eigenschaft oder eine Methode aufrufen möchten.

```
$datum.DayOfWeek
Monday
```

Sie möchten, wissen, der wievielte Tag des Jahres das ist? Kein Problem!

```
$datum.DayOfYear
215
```

Aber mit den Methoden, die Ihnen ebenfalls zur Verfügung stehen, können Sie noch deutlich mehr erreichen, denn Methoden sind Programme, die am Objekt hängen und von eine Programmierer bei der Erstellung der Vorlage (man sagt im Fachsprech Klasse) für Datumswerte erstellt wurden. Methoden rufen Sie auf wie Eigenschaften, allerdings hängen Sie zusätzlich Klammern an den Namen, da Methoden auch Übergabeparameter haben können, die dann in den Klammern übergeben werden.

```
$datum.Adddays(1)
Dienstag, 4. August 2015 10:38:19
```

In diesem Fall haben wir die Methode *Adddays()* verwendet, um einen Tag auf das in $Datum gespeicherte Datum aufzurechnen. Womit die berechtigte Frage aufkommt – wofür braucht man das? Ein sehr simples, aber effektives Beispiel ist der *Get-Eventlog*, der Ihnen auch die Möglichkeit bietet, Ereignisse nur aus einem bestimmten Zeitraum anzuzeigen – sehr effektiv, wenn man einen bestimmten Fehler sucht, der zu einem bestimmten Datum aufgetreten ist. Hierfür verwenden Sie die beiden Parameter *-before* und *-after*:

```
$datum = Get-Date
$vor = $datum.Adddays(-14)
$nach = $datum.Adddays(-28)
Get-eventlog –LogName System –EntryType Error –before $vor –After $nach
```

Wenn Sie genau wissen wollen, welche Parameter die Methode *Adddays()* verwendet, rufen Sie die Methode ohne Klammern auf – Powershell zeigt Ihnen dann, ob und wenn ja, welche Parameter Sie übergeben können:

```
$datum.adddays

OverloadDefinitions
-------------------
datetime AddDays(double value)
```

Die OverloadDefinitions sagen Ihnen, dass AddDays genau 1 Parameter benötigt – nicht mehr, aber auch nicht weniger. Die Anzahl der Tage, die übergeben werden, ist also ein Pflichtparameter. Im deutschen spricht man von Überladungen. Überladung bedeutet, dass die Methode verschiedene Möglichkeiten anbietet, Parameter zu übergeben. Je nachdem, wie viele oder welche Parameter Sie angeben, verändert sich das Verhalten der Methode. Schauen Sie sich dazu einmal die Methode

Substring() an, die jeder String zur Verfügung stellt und der es Ihnen ermöglicht, aus einem bestehenden Text einen Teil auszuschneiden.

```
$text.Substring

OverloadDefinitions
-------------------
string Substring(int startIndex)
string Substring(int startIndex, int length)
```

Substring gibt uns Überladungen. Entweder ruft man Substring nur mit einem Parameter auf – hier wird ein Int erwartet, also eine Ganzzahl:

```
$text.Substring(7)

Voges
```

Unsere Text-Variable enthält den Text Holger Voges. In der ersten Version mit einem Parameter liefert Substring ab dem 8. Zeichen (die Methode beginnt von 0 an zu zählen) den Rest des Strings zurück. Wollen wir aber z.B. nur dem Vornamen zurück liefern, müssen wir auch angeben, wie viele Zeichen wir haben möchten. Das können wir mit der 2. Überladung erreichen:

```
$text.Substring(0,6)

Holger
```

Methoden sind also kleine Helferlein, die man sich auch selber programmieren könnte, wenn man fit an der Tastatur ist – aber warum das Rad 2 Mal erfinden, wenn uns schon jemand die Arbeit abgenommen hat? Jetzt bleibt nur noch die Frage offen, welche Methoden uns so ein Objekt denn zur Verfügung stellt. *Select-Object –Property ** zeigt nur Eigenschaften an. Auch hierfür sorgt Powershell vor. Mit dem Cmdlet *Get-Member* können Sie sich anzeigen, was ein Objekt so alles mitbringt:

```
$datum | Get-Member


   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         datetime Add(timespan value)
AddDays              Method         datetime AddDays(double value)
AddHours             Method         datetime AddHours(double value)
AddMilliseconds      Method         datetime AddMilliseconds(double value)
AddMinutes           Method         datetime AddMinutes(double value)
AddMonths            Method         datetime AddMonths(int months)
[…]
```

Damit wird der Trick mit dem *Select-Object –Property ** (oder kurz *Select **) aber nicht obsolet. Denn *Get-Member* zeigt Ihnen zwar alle Methoden und Eigenschaften, aber nicht, welche Werte in den Eigenschaften stehen. Wollen Sie also nur die Eigenschaften eines Objekts erkunden, nutzen Sie am besten *Select-Object –Property **, wollen Sie die Methoden sehen, nutzen *Sie Get-Member –MemberType Method*.

```
$datum | Get-Member –MemberType Method
```

Arbeiten mit der Pipeline

Die Pipeline ist ein Konstrukt in Powershell, dass Ihnen vor allem bei der Arbeit in der Konsole viel Zeit sparen kann, denn mit Hilfe der Pipeline können Sie eine große Anzahl von Objekte in einem Rutsch bearbeiten. Die Pipeline arbeitet ähnlich einem Fließband. Der erste Befehl in der Pipeline gibt Daten aus, wie z.B. Events aus dem Eventlog, eine Eingabe aus einer Textdatei oder Dateien auf der Festplatte. Mit Hilfe des | Symbols (senkrechter Strich, rechts neben der Linken Shift-Taste) können Sie Daten jetzt zur Weiterverarbeitung an den nächsten Befehl übergeben, und zwar Objekt für Objekt. Die Pipeline kann dabei, genau wie ein Fließband, beliebig viele "Arbeiter" einstezen, die Ihre Daten weiterverarbeiten. Eine sehr einfache Verwendung der Pipeline haben wir dabei bei der Ausgabe schon kennen gelernt:

```
Get-Eventlog –Logname Application | Out-Gridview
```

Hier werden alle Events aus dem Anwendungslog von *Get-Eventlog* ausgelesen, dann an *Out-Gridview* weitergeleitet, aufgehübscht und in einem Fenster ausgegeben. Während *Get-Eventlog* spezialisiert ist auf das Auslesen und Filtern des Eventlogs, kann *Out-Gridview* beliebige Daten visualisieren.

Es gibt eine Reihe von Cmdlets, die auf die Verarbeitung von Daten in der Pipeline spezialisiert sind. Die wichtigsten sind *Select-Object* und *Where-Object*, die beide an die SQL-Abfragesprache angelehnt sind. Während *Select-Object* einzelne Spalten (in der Powershell Eigenschaften oder Properties, die aber vom *Out-Gridview* auch in Spaltenform dargestellt werden) filtern kann, ist der *Where-Object* für das Ausfiltern von bestimmten Datensätzen verantwortlich. Nehmen wir uns hierfür als Beispiel das Arbeiten mit Dateien vor.

Das Cmdlet zum Ausgeben von Dateien heißt *Get-Childitem* – man kann aber auch einfach den Dir-Befehl verwenden, der einfach nur ein Alias ist, das auf den *Get-Childitem* verweist. Zum Aufrufen aller Dateien aus dem Windows-Verzeichnis gibt man einfach ein:

```
Dir C:\Windows
```

Dieser Befehl listet allerdings auch Ordner auf. Um die auszufiltern, gibt es den Parameter *–File* (ab Powershell 3.0):

```
Dir C:\Windows –File


    Verzeichnis: C:\Windows


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---         22.08.2013     13:21      56832 bfsvc.exe
-a--s         31.07.2015     10:51      67584 bootstat.dat
-a---         25.02.2015     09:59       1566 CrmClient.mif
[…]
```

Die Ausgabe listet alle Dateien auf, allerdings nur eine Untermenge der Eigenschaften, die Windows tatsächlich zur Verfügung stellt. Um alle Eigenschaften der Dateien sehen zu können, kann man *Select-Object* verwenden, dem man mit dem Parameter *-Property* angeben kann, welche Eigenschaften er ausgeben soll. Genau wie bei SQL kann man * als Platzhalter für alle Eigenschaften verwenden.

Nun stellt sich die Situation ein wenig anders dar – man bekommt plötzlich keine Tabellarische Ansicht mehr, sondern eine Listenansicht, und es werden deutlich mehr Parameter sichtbar. Um die

Ausgabe ein wenig einzuschränken, kann man den *Select-Object* zusätzlich mit dem Parameter -First verwenden und die Anzahl der Objekte angeben, die ausgegeben werden sollen:

```
Dir C:\Windows -File | Select-Object -Property * -First 1

PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Windows\bfsvc.exe
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSChildName       : bfsvc.exe
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : False
VersionInfo       : File:             C:\Windows\bfsvc.exe
                    InternalName:     bfsvc.exe
                    OriginalFilename: bfsvc.exe.mui
                    FileVersion:      6.3.9600.16384 (winblue_rtm.130821-1623)
                    FileDeskription:  Startdatei-Wartungshilfsprogramm
                    Product:          Betriebssystem Microsoft? Windows?
                    ProductVersion:   6.3.9600.16384
                    Debug:            False
                    Patched:          False
                    PreRelease:       False
                    PrivateBuild:     False
                    SpecialBuild:     False
                    Language:         Deutsch (Deutschland)

BaseName          : bfsvc
Mode              : -a---
Name              : bfsvc.exe
Length            : 56832
DirectoryName     : C:\Windows
Directory         : C:\Windows
IsReadOnly        : False
Exists            : True
FullName          : C:\Windows\bfsvc.exe
Extension         : .exe
CreationTime      : 22.08.2013 13:21:53
CreationTimeUtc   : 22.08.2013 11:21:53
LastAccessTime    : 22.08.2013 13:21:53
LastAccessTimeUtc : 22.08.2013 11:21:53
LastWriteTime     : 22.08.2013 13:21:47
LastWriteTimeUtc  : 22.08.2013 11:21:47
Attributes        : Archive
```

Mit *Select-Object* kann man aber auch die Ausgabe auf die Werte einschränken, die man wirklich benötigt:

```
Dir C:\Windows -File | Select-Object -Property fullname,Length

FullName                                                                Length
--------                                                                ------
C:\Windows\bfsvc.exe                                                     56832
C:\Windows\bootstat.dat                                                  67584
C:\Windows\CrmClient.mif                                                  1566
[…]
```

Jetzt werden nur noch der volle Pfadname und die Dateigröße ausgegeben. Sehr praktisch, wenn man die Daten z.B. per *export-csv* ausgeben möchte.

Will man jetzt aber nur die Dateien sehen, die größer als 1 MB sind, hilft einem *Select-Object* nicht mehr weiter, denn *Select-Object* filtert ja nur Spalten, aber keine einzelnen Datensätze. In SQL ist hierfür das *Where*-Statement notwendig. Entsprechend nennt sich das Powershell-Cmdlet *Where-Object*.

Where-Object ist das Aschenputtel-Cmdlet: Die guten in Töpfchen, die schlechten ins Kröfpchen. Anhand einer Bedingung werden die "schlechten" Objekte vom "Fließband" entfernt. Es ist also die Qualitätskontrolle. Der Filter wird hinter den Where-Object in geschweiften Klammern geschrieben.

Where-object { Objekt.<Objekteigenschaft> Vergleichsoperator <Vergleichswert> }

Der Vergleichsoperator legt fest, welchem Wert die Objekteigenschaft entsprechen (oder nicht entsprechen) soll, damit das Objekt in der Pipeline weiter geleitet wird. Es gibt eine ganze Reihe von Vergleichoperatoren:

Operator	Bedeutet	Entspricht mathematisch / SQL
-eq	Equal (gleich)	=
-ne	Not equal (ungleich)	<>
-ceq	Equal (Groß-Kleinschreibesensitiv)	=
-lt	Lesser Then (kleiner als)	<
-le	Lesser or equal (kleiner gleich)	<=
-gt	Greater Then (größer als)	>
-ge	Greater or Equal (größer gleich)	>=
-like	Wie	Like
-contains	Beinhaltet	IN

Für eine Auflistung aller Vergleichsoperatoren verwenden Sie *get-Help about_comparison_operators*.

Um alle Dateien zu filtern, die größer oder gleich 1 MB sind, würde der Befehl entsprechend lauten:

```
Dir C:\Windows -File | Where-Object { $_.Length -ge 1MB }
```

Interessant an dieser Stelle ist zum einen das *$_* im Filter. *$_* gehört quasi zur Syntax des Where-Object. *$_* beinhaltet innerhalb der geschweiften Klammern immer das Objekt, das gerade in der Pipeline verarbeitet wird. Es ist einfach ein Platzhalter, in der sich die jeweils aktuelle Datei befindet. *.Length* ist dann jeweils die Größe-Eigenschaft der Datei. Darauf folgen der Vergleichsoperator und dann der Vergleichswert.

Powershell kann Größenangaben direkt umrechnen, indem Sie einfach die Einheit direkt ohne Leerzeichen hinter die Zahl schreiben. So können Sie z.B. auch 10MB schreiben oder 100KB oder 13GB. Als Ausgabe erhalten Sie alle Dateien, die größer als 1 MB sind. Nun können Sie wieder den Select-Object verwenden, um die Ausgabe auf den Pfad und die Größe einzuschränken:

```
Dir C:\Windows -File | Where-Object { $_.Length -ge 1MB } | Select-Object -Property
fullname,Length

FullName                                                                  Length
--------                                                                  ------
C:\Windows\explorer.exe                                                  2501368
C:\Windows\MEMORY.DMP                                                 1148838462
C:\Windows\ntbtlog.txt                                                   2370268
C:\Windows\Procmon.pmb                                                79622594980
C:\Windows\WindowsUpdate.log                                             2087678
```

Natürlich können Sie die Ausgabe jetzt auch weiter an Out-Gridview leiten. Prinzipiell setzt hier nur Ihre Phantasie Grenzen.

Interessant beim Arbeiten mit Vergleichsoperatoren ist, dass Sie tatsächlich auch ohne den Where-Object wunderbar arbeiten. Geben Sie z.B. in die Kommandozeile ein

```
10 -gt 100

False
```

sagt Ihnen Powershell, dass das wohl nicht ganz richtig sein kann. Der *Where-Object* ist also eigentlich recht stumpf – eigentlich wertet Powershell einfach für jedes Objekt in der Pipeline die Bedingung aus, und Where-Object filtert alles aus, wo Powershell *false* ausgibt. Das Ergebnis der Überprüfung muss dementsprechend also immer wahr oder falsch sein. Dies hat aber einen tollen Nebeneffekt. Sie können mit dem Where-Object nämlich auch phantastisch auf das Vorhandensein von Eigenschaften filtern. Suchen Sie z.B. alle Benutzer im Active Directory, bei denen die Eigenschaft *City* gesetzt ist, können Sie einfach alle Benutzer abrufen und über den Where-Object die Benutzer herausfiltern, für die die Eigenschaft gesetzt ist. Beachten Sie bitte, dass für diesen Befehl das AD-Modul installiert sein muss – mehr dazu im Kapitel AD verwalten mit Powershell.

```
Get-Aduser –Filter * | where-object { $_.City }
```

Ist die Eigenschaft gesetzt, liefert Powershell *true* zurück und *Where-Object* leitet das Objekt weiter, ist keine Eigenschaft gesetzt, wird das Objekt ausgefiltert. Wenn Sie die Suche umkehren möchten, also alle Benutzer ausgeben wollen, die in der Eigenschaft *City* nichts eingetragen haben, können Sie mit dem Operator *-not* das Ergebnis einfach umdrehen, aus wahr wird dann falsch und aus falsch wird wahr.

```
Get-Aduser | where-object { -not ($_.City) }
```

Arbeiten mit Skripten

Ein großer Vorteil bei der Arbeit mit der Powershell ist, dass Sie auch Skripte erstellen können. Mit Skripten können Sie ihre Kommandos immer wieder starten, ohne sie jedes Mal komplett in die Powershell eingeben zu müssen. Faktisch ist ein Skript einfach nur eine Reihe von Powershell-Befehlen, die Sie in eine Textdatei mit der Endung ps1 speichern. Zum Schreiben von Skripten hilft Ihnen dabei die ISE (Integrated Skripting Environment), ein seit Powershell 2.0 eingebauter Editor, der das Schreiben von Skripten deutlich vereinfacht.

Um die ISE zu starten, geben Sie entweder ISE in eine geöffnete Powershell-Konsole ein oder, wenn vorhanden, öffnen Sie das Kontextmenü der Powershell in der Taskleiste und wählen Sie Windows

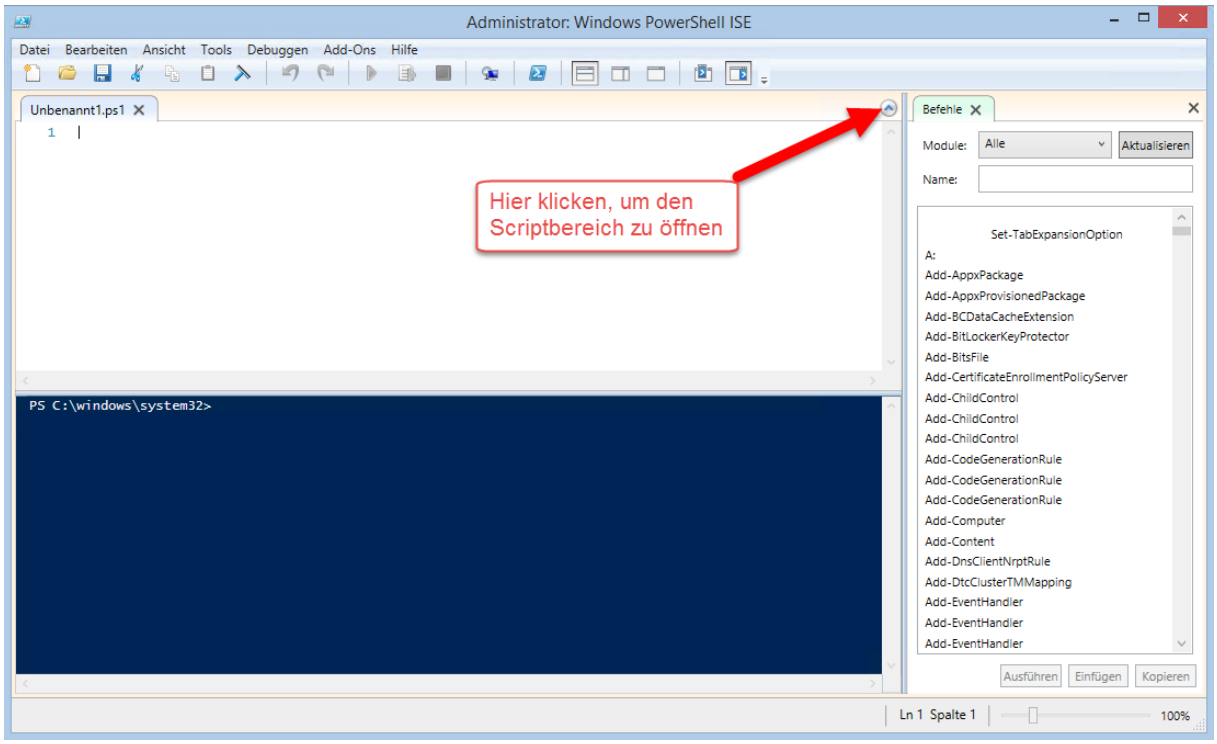


Abbildung 4 - Die ISE 2.0

Powershell ISE. Daraufhin sollte sich die ISE öffnen. Auf einem Windows Server ist die ISE allerdings nicht standardmäßig vorhanden, sondern muss als Feature erst nachinstalliert werden. Das geht über den Server-Manager oder ganz einfach mit folgendem Powershell-Befehl:

```
Import-Module –Name Servermanager  # nur mit Powershell 2.0 notwendig
Add-WindowsFeature –Name PowerShell-ISE
```

Je nachdem, welche Version der Powershell Sie im Einsatz haben, wird sich die Ansicht bei Ihnen eventuell unterscheiden. Die ISE 1.0, die mit Powershell 2.0 mitgeliefert wurde und bei Windows 7 Standard ist, ist von der Funktionalität noch deutlich rudimentärer.

Wenn Sie die ISE das erste Mal starten, sehen Sie nur eine Menüleiste, das Befehls-Add-on auf der rechten Seite und das blaue Fenster mit der Eingabeaufforderung. Um ein Skript zu erstellen, können Sie entweder das erste Icon oben Links anklicken, oder den kleinen Pfeil, der sich links neben dem Befehls-Add-on befindet (s. Abbildung). Dann öffnet sich ein Eingabe-Tab, der Ihnen das Erstellen von Skripten ermöglicht. Geben Sie hier einfach einen Powershell-Befehl ein – sofort öffnet sich ein Fenster, das Ihnen alle Cmdlets anzeigt, die mit der Zeichenkombination beginnen, die Sie bereits getippt haben. Sie können so mit der Maus oder mit den Pfeiltasten das richtige Cmdlet auswählen und mit der Tabulator-Taste bestätigen. Das Kommando wird dann in das Skript eingefügt. Dieses Feature funktioniert im Übrigen auch, wenn Sie in die blaue Eingabeaufforderung wechseln und nennt sich Intellisense.

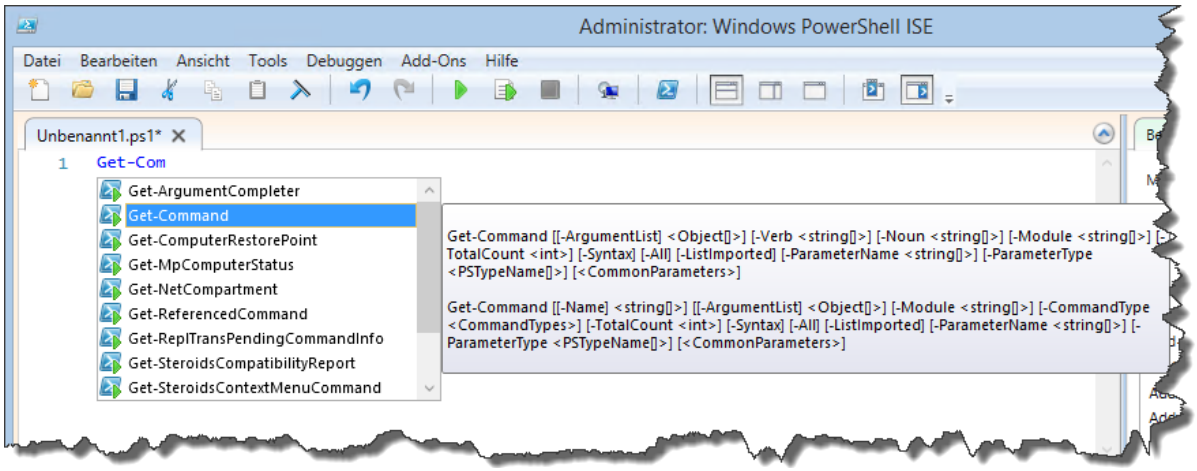


Abbildung 5 - Intellisense: Alles Cmdlets mit Get und com im Namen werden angezeigt, inkl. der Parameter

Die ISE kann aber auch mehr. Wenn Sie ein Skript geschrieben haben, können Sie dieses direkt mit F5 ausführen. Alternativ können Sie auch den grünen Play-Button in der Icon-Leiste auswählen. Der Button direkt rechts neben dem Play-Button führt nur entweder die aktuelle Auswahl aus, oder die Zeile, in der sich der Cursor gerade befindet. Das gleich Verhalten erreichen Sie auch mit der F8-Taste.

Schreiben Sie ein Skript und speichern es zum ersten Mal, kann es Ihnen passieren, dass Sie das Skript hinterher nicht mehr ausführen können. Stattdessen erklärt Ihnen die Powershell, dass die Ausführung von Skripten verboten ist.

```
Die Datei "C:\temp\test.ps1" kann nicht geladen werden, da die Ausführung von
Skripts auf diesem System
deaktiviert ist.
```

Dies ist ein spezielles Sicherheitsfeature von Powershell, die sogenannte Ausführungsrichtlinie. Ausführungsrichtlinien sollen vermeiden, dass jedes x-beliebige Skript einfach ausgeführt werden kann. Die Ausführungsrichtlinie verhindert nicht, dass man Befehle interaktiv in der Powershell-Konsole ausführen kann, es verhindert lediglich das Starten von Skripten. Die Ausführungsrichtlinie kann aber ebenfalls über Powershell sehr leicht konfiguriert werden. Dafür steht Ihnen der Befehl Set-Executionpolicy zur Verfügung. Powershell kennt mehrere Ausführungsrichtlinien:

Richtlinie	Auswirkung
Restricted	Es werden keine Skripte ausgeführt
AllSigned	Alle Skripte müssen über eine digitale Signatur verfügen
RemoteSigned	Skripte,die aus dem Netzwerk gestartet werden, benötigen eine digitale Signatur. Lokale Skripte können gestartet werden. Dies ist in den meisten Fällen die empfohlene Richtlinie
Unrestricted	Skripte werden (fast) immer ausgeführt
Bypass	Skripte werden immer ausgeführt. Der Unterschied zu Unrestricted besteht darin, dass Skript, die aus dem Internet heruntergeladen wurden und als potentiell gefährlich markiert sind, auch im Modus unrestricted nicht ausgeführt werden.

Die Ausführungsrichtlinie kann nicht nur in der Shell selber durch den Administrator, sondern auch durch eine Gruppenrichtlinie konfiguriert werden. Dabei überschreiben Gruppenrichtlinien immer die lokalen Einstellungen. Außerdem kann beim Starten der Powershell.exe über den Parameter *-Executionpolicy* die lokal eingestellte Richtlinie ebenfalls überschrieben werden. Welche Richtlinie in Ihrer Konsole gerade gilt, können Sie über den Parameter *-list* im Cmdlet *Get-Executionpolicy* ganz einfach überprüfen.

```
Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    Unrestricted
```

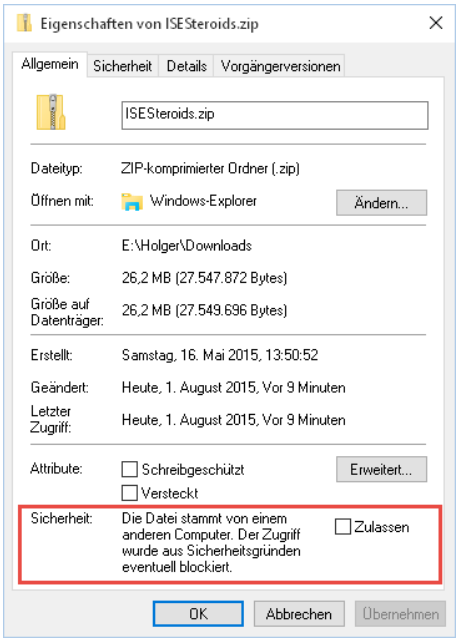


Abbildung 6- Ausführbare Dateien werden geblockt

Die Richtlinien sind in der Anzeige in Ihrer Priorität aufgelistet. *LocalMachine* ist die Einstellung, die für alle Benutzer gilt, *CurrentUser* ist die Einstellung, die für den aktuellen Benutzer gesetzt wurde, *Process* ist die Richtlinie, die über die Powershell.exe gesetzt wurde, *UserPolicy* ist die Einstellung, die in den Benutzereinstellungen einer Gruppenrichtlinie definiert ist, und *MachinePolicy* ist über die Computerrichtlinie gesetzt wurde. Damit sieht man, dass man über die Powershell.exe jederzeit die Einstellungen überschreiben kann, die in der Konsole explizit angefordert wurden, aber Gruppenrichtlinien sind in der Lage, die Ausführung von Skripten generell zu überschreiben, völlig unabhängig von den lokalen Einstellungen. Um die Einstellungen für den Benutzer oder die Maschine explizit zu setzen, stellt Ihnen *Set-Executionpolicy* übrigens den Parameter *-Scope* zur Verfügung.

Geben Sie sich jetzt aber nicht der Illusion hin, dass durch eine Einstellung in der Gruppenrichtlinie Powershell-Skripte komplett deaktiviert werden können. Tatsächlich ist es sogar für einen Benutzer ohne Administrator-Rechte möglich, diese Einstellungen außer Kraft zu setzen. Am besten sehen Sie die Ausführungsrichtlinie als eine erste Hürde, wenn jemand z.B. über eine mail versucht, einem unbedarften Benutzer ein Skript unterzujubeln. Ein Benutzer mit Hintergrundwissen ist aber in der Lage, diese Sperre zu umgehen.

Womit wir bei einem anderen Thema wären – wie führt man Powershell-Skripte jetzt überhaut aus? Anders als bei .cmd und .bat-Dateien werden .ps1-Dateien nämlich nicht automatisch ausgeführt,

wenn man sie doppelt anklickt. Stattdessen ist die Endung .ps1 mit dem Notepad verknüpft und die Datei wird im Editor geöffnet. Möchten Sie ein Powershell-Skript ausführen, stehen Ihnen eine Reihe von anderen Möglichkeiten zur Verfügung:

- Öffnen Sie das Kontextmenü des Skripts und wählen Sie "Mit Powershell ausführen"
- Starten Sie das Skript in einer Powershell-Konsole. Geben Sie hierfür den kompletten Pfad an oder, wenn sich das Skript im aktuellen Ordner befindet, ".\Skriptname". Powershell führt Skripte nur mit vollqualifiziertem Pfad aus. Nutzen Sie hierfür einfach den Tabulator, er fügt automatisch .\ vor dem Skriptnamen ein
- Starten Sie das Skript über einen Parameter der Powershell.exe. Sie können Powershell.exe entweder mit einem Skript starten (Parameter *-file*) oder direkt ein Kommando ausführen lassen (Parameter *-command*).
- Starten Sie das Skript über einen automatisierten Job (ScheduledJob). Dies ist ein spezieller geplanter Task, der aber nur Powershell-Skripte startet und ab Powershell 3.0 verfügbar ist.
- Starten Sie das Skript z.B. als Login-Skript aus Gruppenrichtlinien heraus.

Um ein Skript über Powershell.exe zu starten, stehen Ihnen eine ganze Reihe von Parameter zur Verfügung. Wenn Sie Powershell.exe /? eingeben, bekommen Sie alle Parameter aufgelistet. Die wichtigsten zur Ausführung von Skripten sind:

Parameter	Funktion
-NoProfile	Lädt keine Profildateien – Profile sind Startskripte ähnlich der autoexec.bat von DOS, die vor der Ausführung Ihres Skriptes gestartet werden und die Umgebung vorkonfigurieren können.
-WindowStyle	Kann ein Fenster auch unsichtbar starten, so dass keine Interaktion mit der Benutzeroberfläche stattfindet
-Executionpolicy	Legt die Ausführungsrichtlinie fest und überschreibt alle lokalen Einstellungen
-Command	Ein oder mehrere Powershell-Kommandos, die ausgeführt werden sollen. Die Kommandos müssen als Skriptblock (in geschweiften Klammern { } angegeben werden
-File	Gibt den Pfad zu einem Skript an. Kann nicht gemeinsam mit –Command genutzt werden.

Um das Skript NewTask.ps1 aus der Freigabe \\FS1\Skripte\ im Hintergrund zu starten und die Standard-Ausführungsrichtlinie zu überschreiben, würde die Kommandozeile also folgendermaßen aussehen:

```
Powershell.exe –NoProfile –ExecutionPolicy Bypass –File \\FS1\Skripte\NewTask.ps1
```

Einfache Skripte mit Übergabeparametern und eingebauter Hilfe

Skripte sollen normalerweise sich wiederholende Aufgaben automatisieren. Viele Werte, die in einem Skript verwendet werden, sind aber variabel, wie z.B. Pfade oder Computernamen. Daher gibt es die Möglichkeit, Powershell-Skripte genau wie Cmdlets mit Parameter auszustatten, die beim Aufrufen des Skriptes angegeben werden. Dadurch können Skripte genau wie Cmdlets aufgerufen

werden, und Sie müssen nicht jedes Mal das Skript öffnen und editieren. Den Einsatz von Parametern möchte ich an einem einfachen Beispiel demonstrieren – dem Erstellen einer geplanten Aufgabe, oder einem scheduled Task, wie es im englischen heißt.

Ein geplanter Task besteht aus einer Reihe von Komponenten, die in Powershell einzeln angelegt werden müssen. Dies sind:

- Ein Trigger oder Auslöser, der definiert, wann die Aufgabe ausgeführt werden soll. Dies kann ein Zeitplan sein, aber auch ein Ereignis (Event) wie der Start des Betriebssystems oder das an- oder abmelden eines Benutzers.
- Eine Aufgabe, die ausgeführt werden soll. Dies kann jedes beliebige ausführbare Programm sein.
- Eine Reihe von Optionen, wie z.B. ob der Task auch ausgeführt werden soll, wenn der Computer sich im Batteriebetrieb befindet.

Für jede dieser Komponenten gibt es ein eigenes Cmdlet, das ein entsprechendes Konfigurationsobjekt erzeugt, dass in einer Variablen gespeichert werden muss.

Zum Erstellen des Zeitplans wird das Cmdlet *New-ScheduledTaskTrigger* verwendet. Wenn Sie täglichen Start konfigurieren möchten, verwenden Sie den Parameter *-Daily*. Daily ist ein Switch-Parameter, benötigt also kein Argument, allerdings den zweiten Parameter *-Daysinterval*, der definiert, in welchem Abstand (in Tagen) der Task gestartet werden soll. Außerdem muss mit dem Parameter *-At* eine Zeit definiert werden. Hier können Sie einfach eine Zeit in englischem Format vorgeben:

```
New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM
```

Wenn Sie einen Ereignisbasierten Start konfigurieren möchten, verwenden Sie statt daily z.B. *-AtStartup*, *-AtLogon* oder ähnliches. Suchen Sie sich die notwendigen Parameter einfach aus der automatischen Parameter-Ergänzung mit der TAB-Taste.

Zum Erstellen der Aktion, die ausgeführt werden soll, verwenden Sie den Parameter *New-ScheduledTaskAction*. Das Cmdlet erwartet vor allem 2 Parameter: *-Execute* gibt das Kommando an, dass Sie ausführen lassen möchten, und *-Argument* übernimmt die Parameter samt Argumenten, die der Befehl erwartet. Möchten Sie ein Powershell-Skript automatisch starten, geben Sie hier also Powershell.exe als auszuführendes Programm an, und mit -Argument übergeben Sie die Parameter wie das Skript, das gestartet werden soll.

```
New-ScheduledTaskAction -Execute "Powershell.exe" –Argument
"-noprofile -executionpolicy bypass -file \\FS1\Skripte\RestartComputer.ps1"
```

Im Beispiel erzeugen Sie einen Task, der ein Skript aus einer Netzwerkfreigabe startet und die Ausführungsrichtlinie für diesen Durchlauf auf bypass setzt.

Nun können Sie den Task registrieren. Das passiert nicht, wie man vermuten könnte, mit dem Cmdlet *New-ScheduledTask*, sondern mit dem Cmdlet *Register-ScheduledTask*. Da beim Registrieren sowohl die Aktion als auch der Zeitplan mit angegeben werden müssen, müssen wir beide Objekte vorher in einer Variablen gespeichert haben. Das fertige Skript würde also folgendermaßen aussehen:

```
$Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM
$Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile  -executionpolicy bypass -file \\FS1\Skripte\RestartComputer.ps1"
Register-ScheduledTask -TaskName "Restart Computer" -Action $Action -Trigger
$Trigger -User "System" -Deskription "Restart Computer"
```

Speichern wir die Befehle als .ps1-Datei, haben wir ein Skript, aber leider ein ziemlich unflexibles. Soll nämlich für den Task ein anderer Name angelegt werden, muss man das Skript öffnen und bearbeiten. Soll eine andere Skriptdatei angeben werden, die gestartet werden soll, gilt das gleich.

Um das Problem zu lösen, ersetzen wir zuerst die Werte, die variabel gehalten werden sollen, durch Variablen. Dann sieht unser Skript so aus:

```
$Filename = "\\FS1\Skripte\RestartComputer.ps1"
$Taskname = "Restart Computer"

$Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM
$Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile  -executionpolicy bypass -file $filename
Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -User
"System" -Deskription "Restart Computer"
```

Das erleichtert jetzt zwar die Wartung des Skripts, aber das sind noch keine Übergabeparameter. Der Rest ist jetzt allerdings sehr einfach. Die Parameter, die über die Kommandozeile übergeben werden sollen, werden nämlich direkt in Variablen übergeben, die heißen wie die Parameter. Alles, was wir dafür tun müssen, ist die erstellen Variablen in einem param()-Block einzuschließen und mit Kommas zu trennen:

```
Param(
   $Filename = "\\FS1\Skripte\RestartComputer.ps1",
   $Taskname = "Restart Computer"
)

$Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM

$Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile  -executionpolicy bypass -file $filename

Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -User
"System" -Deskription "Restart Computer"
```

Achten Sie auf das Komma hinter dem ersten Parameter - alle Parameter müssen durch Kommas voneinander getrennt werden! Wir haben hier außerdem bereits eine Spezialfunktion von Parametern verwendet, nämlich Parameter mit Default-Werten. Den Parametern ist bei der Deklaration bereits mit einem Gleichheitszeichen ein Wert zugewiesen worden. Wenn das Skript mit Parametern aufgerufen wird, wird der Standardwert einfach überschrieben. Geben Sie den Parameter beim Aufruf des Skripts nicht an, wird der im Skript definierte Standard verwendet.

Speichern Sie das Skript jetzt, z.B. unter dem Namen "Add-Task.ps1". Um es aufzurufen, geben Sie die Parameter einfach an, als würden Sie ein Cmdlet aufrufen, also mit Bindestrich. Powershell zeigt Ihnen die Parameter sogar per Kommandozeilenerweiterung an:

```
.\Add-Task –Filename "\\FS1\Skripte\GetComputerinfo.ps1" –Taskname "Inventory"
```

Das Skript lässt sich jetzt flexible auf unterschiedliche Tasks anpassen. Versuchen Sie doch einfach mal, auch die Auszuführende Aufgabe als Parameter zu hinterlegen.

Skripte Dokumentieren

Powershell stellt Ihnen auch die Möglichkeit zur Verfügung, Kommentare zu Ihrem Skript hinzuzufügen. Kommentare sind Text, der von Powershell ignoriert wird. Hierfür stehen Ihnen zwei Möglichkeiten zur Verfügung:

```
# (Raute, Lattenzaun, Hashtag): Kommentiert alles aus, was hinter dem # steht

<#
Dies ist ein
```

```
Mehrzeiliger Kommentar oder
Blockkommentar
#>
```

Die einfache Raute kommentiert alles aus, was in derselben Zeile hinter dem # steht. Um mehrere Zeilen als Kommentar zu markieren, stellen Sie an den Start des Kommentars <# und ans Ende #> - echt simpel.
Ich möchte mich an dieser Stelle nicht darüber ergehen, wie wichtig es ist, Ihren Code zu kommentieren. Sie sollten auf jeden Fall an der einen oder anderen Stelle im Code Kommentare erzeugen, schon allein, um Ihren eigenen Code nach einiger Zeit selber noch zu verstehen. Ich möchte Ihnen die stattdessen zeigen, wie Sie mit einfachen Kommentaren in der Lage sind, Ihr Skript mit einer Onlinehilfe zu versehen, die Ihr Skript über *get-Help* genauso kommentiert wie ein Cmdlet. Am einfachsten verwenden Sie hierfür ein sogenanntes Snippet als Vorlage. Snippets sind Codeschnipsel, die Sie in der ISE mit der Tastenkombination Strg+J erreichen. Klicken Sie einfach irgendwo in Ihr Skript und drücken Sie Strg+J, und es öffnen sich eine Reihe von Standard-Snippets. Wählen Sie hier das erste Snippet *Cmdlet (erweiterte Funktionen)* aus.

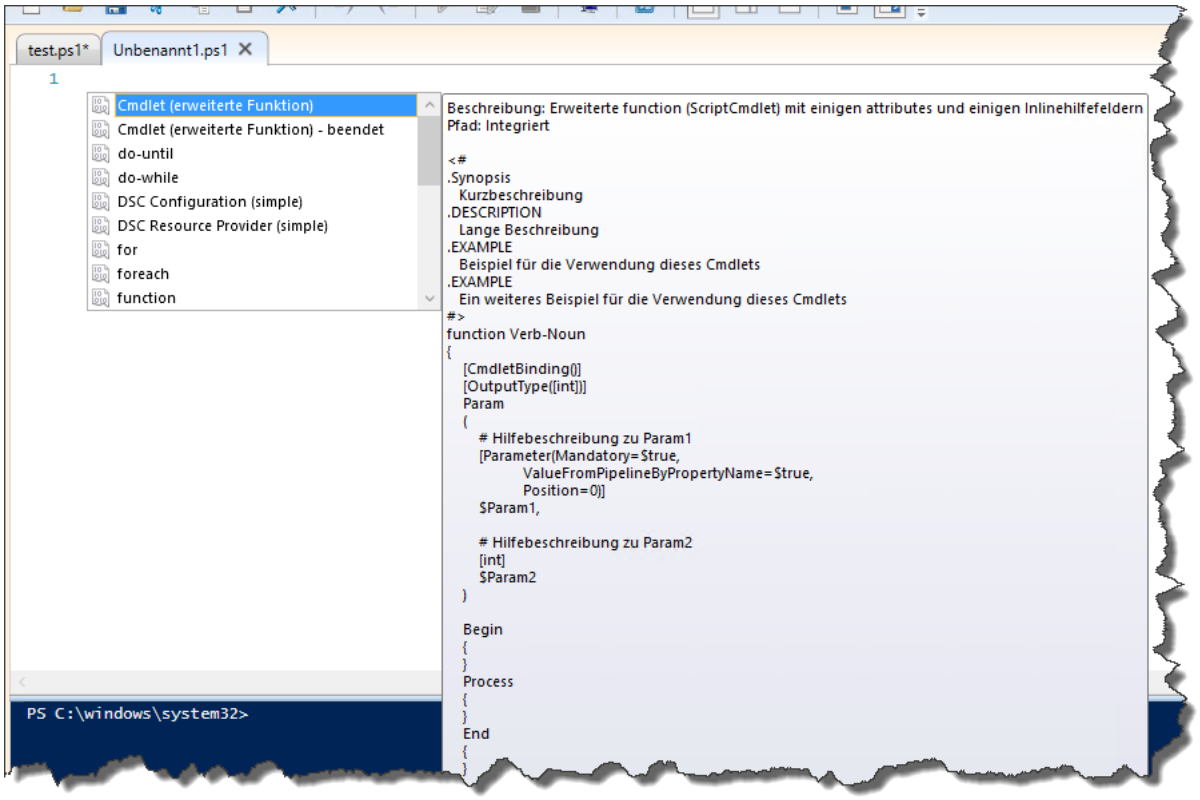


Abbildung 7 - Der Snippet Manager stellt eine Reihe von Code-Schnipseln zur Verfügung

Ich möchte nicht auf alle Funktionalitäten der erweiterten Funktionen eingehen – Wenn Sie mehr wissen möchten, suchen Sie einfach in der Powershell-Hilfe nach "about_functions-advanced". Interessant ist hier jedoch der Teil, der am Anfang des Skripts über dem Befehl "function" steht. Es handelt sich um einen Blockkommentar, den Sie einfach per Copy and Paste in Ihr Skript über den param-Block kopieren können. Tragen Sie jetzt unter .Synopsis in Kurzform ein, was Ihr Skript tut, unter .Deskription eine ausführlichere Beschreibung, und unter .Example können Sie Beispiele eintragen. Wenn Ihnen 2 Beispiele nicht reichen, fügen Sie einfach weitere .Example-Zeilen ein. Zusätzlich können sie im Param-Block noch mit einem einfachen Zeilenkommentar eine kurze Beschreibung der Parameter einfügen:

```
<#
.Synopsis
```

```
   Skript zum Anlegen von geplanten Tasks
.DESCRIPTION
   Legt einen geplanten Task an, der täglich um 1 Uhr läuft und ein Powershell-
Skript startet
.EXAMPLE
   .\Add-Task
   Legt einen neuen Task mit Standardwerten an
.EXAMPLE
   .\Add-Task -Filename "\\FS1\Skripte\GetComputerinfo.ps1" -Taskname "Inventory"
   Erzeugt einen neuen Task mit Namen Inventory, der das Skript GetComputerinfo
   Startet.
#>
Param(
   # Der Pfad der Skriptdatei, die automatisch gestartet werden soll
   $Filename = "\\FS1\Skript\RestartComputer.ps1"
   # Der Name, der für den geplanten Task vergeben werden soll
   $Taskname = "Restart-Computer"
)

$Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM
$Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile   -executionpolicy bypass -file $filename
Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -User
"System" -Deskription "Restart Computer"
```

Speichern Sie nun Ihr Skript ab, rufen Get-help mit Ihrem Skriptnamen auf und staunen Sie!

```
Get-help -examples

NAME
    C:\temp\add-task.ps1

ÜBERSICHT
    Skript zum Anlegen von geplanten Tasks

    -------------------------- BEISPIEL 1 --------------------------

    C:\PS>.\Add-Task

    Legt einen neuen Task mit Standardwerten an
    -------------------------- BEISPIEL 2 --------------------------

    C:\PS>.\Add-Task -Filename "\\FS1\Skripte\GetComputerinfo.ps1" -Taskname
"Inventory"

    Erzeugt einen neuen Task mit Namen Inventory, der das Skript GetComputerinfo
    Startet.
```

Sollte die Ausgabe der Hilfe nicht klappen, überprüfen Sie einmal in der ISE, ob irgendwo Skriptfehler angezeigt werden (rote Schlangenlinie) und wie Ihre Ausführungsrichtlinie eingestellt ist.

Powershell Remoting

Mit Powershell 2.0 hat Microsoft der Powershell eine neue Funktion spendiert, die das Steuern mehreren Computern gleichzeitig wesentlich vereinfacht – das Powershell Remoting. Mit Remoting ist es möglich, sich über die Powershell mit jedem Computer im Netzwerk zu verbinden und Skripte auszuführen. Um das Remoting nutzen zu können, benötigt man den Dienst "Windows Remotverwaltung", der ab dem Microsoft Management Framework 2.0 automatisch installiert wird. Ab Windows Server 2008 R2 ist der Dienst auch automatisch eingeschaltet, auf Windows Clients muß er erst aktiviert werden. Dafür nutzen Sie entweder das Kommandozeilentool winrm.exe oder das Cmdlet *Enable-PSRemoting*.

Um das Remoting zu verwenden, stehen Ihnen grundsätzlich 2 Möglichkeiten zur Verfügung. Entweder verbinden Sie sich direkt mit dem zu konfigurierenden Server. Hierfür nutzen Sie das Cmdlet *Enter-PSSession*.

```
Enter-PSSession –Computername FS1
```

Mit diesem Cmdlet wird sofort eine Verbindung mit dem Server hergestellt. Geben Sie keine weiteren Anmeldeinformationen an, werden Sie mit dem Namen und Kennwort des Benutzers verbunden, mit denen Sie gerade angemeldet sind (Single Sign on). Alternativ können Sie über den Parameter *-Credential* ein anderes Benutzerkonto angeben.

```
Enter-PSSession –Computername FS1 –Credential Get-Credential
```

Sie werden direkt mit dem Zielcomputer verbunden. Dass die Verbindung erfolgreich war, sehen Sie am Servernamen im Prompt der Powershell. Wenn Sie die Sitzung verlassen möchten, geben Sie einfach *exit* ein. Voraussetzung, dass die Verbindung problemlos klappt, ist allerdings eine Windows-Domäne. Das Windows-Anmeldeprotokoll Kerberos kann dann die Absicherung übernehmen. Außerhalb einer Domäne sind eine Reihe von Konfigurationseinstellungen vorzunehmen, damit die Anmeldung erfolgreich durchgeführt werden kann. Außerdem müssen die tcp-Ports 5985 und 5986 auf dem Zielsystem erreichbar sein, da Powershell Remoting diese Ports für die Kommunikation verwendet.

Port	Funktion
Tcp, 5985	Unverschlüsselte Kommunikation
Tcp, 5986	Verschlüsselter Datenaustausch über https

Eine ausführliche Beschreibung, wie Sie Powershell Remoting außerhalb einer Domäne konfigurieren müssen, finden Sie im kostenlosen ebook "Secrets of Powershell Remoting", dass Sie bei Penflip.com herunterladen können.

Noch interessanter als *Enter-PSSession* ist das Cmdlet *Invoke-Command*. Invoke heißt so viel wie Aufrufen und startet ein oder mehrere Kommandos oder Skripte auf dem Zielsystem, ohne dass Sie sich interaktiv mit der Konsole verbinden müssen. Außerdem beherrscht *Invoke-Command* ein Feature, das sich Ausfächern nennt. Beim Ausfächern geben Sie einfach mehrere Computer an, auf denen die Kommandos ausgeführt werden sollen. Invoke-Command verbindet sich dann selbständig mit den aufgelisteten Computern und führt die Befehle dort aus. Möchten Sie beispielsweise eine Gruppe von Computern neu starten, genügt folgende Kommandozeile:

```
Invoke-Command –Computername SQL1,SQL2,SQL3 –Skriptblock { Restart-Computer }
```

Achten Sie darauf, dass das Kommando hinter dem Parameter in geschweiften Klammern steht, also als Skriptblock angegeben ist. Wenn Sie mehr als ein Kommando absetzen möchten, haben Sie zwei Möglichkeiten:

- Trennen Sie die Befehle voneinander mit einem Semikolon
- Innerhalb eines Skript-Blocks können Sie die Befehle wieder mit einem Enter auf mehrere Zeilen verteilen.

Möchten Sie also vor dem Herunterfahren erst einen Dienst stoppen und eine Datei ins Netzwerk kopieren, können Sie das mit folgendem Befehl realisieren:

```
Invoke-Command –Computername SQL1,SQL2,SQL3 –Skriptblock {
   Stop-Service –Name MSSQL
   Copy-Item -Path C:\backup\sqlbackup.bak –Destination \\fs1\backup\sqlbackup.bak
   Restart-Computer
   }
```

Dies ist gegenüber der Semikolon-getrennten Variante die deutlich übersichtlichere.

Mit dem Parameter *-Filepath* kann man statt eines Skriptblocks auch ein Skript übergeben. Lassen Sie den Parameter Skriptblock einfach weg, und geben Sie stattdessen mit *-Filepath* den Pfad zum Skript an:

```
Invoke-Command –Computername SQL1,SQL2,SQL3 –FilePath C:\skripte\sqlbackp.ps1
```

Im Folgenden erweitern wir das oben erstellte Skript zum Erstellen von geplanten Tasks um die Funktion, auf beliebigen Computern geplante Tasks anzulegen. Alles, was wir dafür tun müssen, ist unserem Skript einen zusätzlichen Parameter *Computername* hinzuzufügen und die Befehle zum Anlegen des Tasks mit *Invoke-Command* zu umschließen:

```
Param(
   # Der Pfad der Skriptdatei, die automatisch gestartet werden soll
   $Filename = \\FS1\Skript\RestartComputer.ps1

   # Der Name, der für den geplanten Task vergeben werden soll
   $Taskname = "Restart-Computer",

   # Der Name der Computer, auf denen ein Task angelegt werden soll
   $Computername
)

Invoke-Command –Computername $Computername –Skriptblock {
  $Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM

  $Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile   -executionpolicy bypass -file $filename

  Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -
User "System" -Deskription "Restart Computer"

  }
```

Wir können sogar noch weiter gehen und eine Liste von Computern als Textdatei angeben. Dafür legen wir einen weiteren Parameter *–Computerlist* an. Mit *Get-Content* lesen wir die Datei aus dem Pfad ein. In der Textdatei müssen die Computernamen in jeweils einer eigenen Zeile stehen, damit das Kommando funktioniert. Nach dem Einlesen müssen wir die Liste der eingelesenen Computer und die mit *-Computername* angegebenen Computer in einer Variablen addieren. Das geht in Powershell ganz einfach mit dem Plus-Zeichen (+).

```
Param(
   # Der Pfad der Skriptdatei, die automatisch gestartet werden soll
   $Filename = \\FS1\Skript\RestartComputer.ps1

   # Der Name, der für den geplanten Task vergeben werden soll
   $Taskname = "Restart-Computer",

   # Der Name der Computer, auf denen ein Task angelegt werden soll
   $Computername,

   # Eine Textdatei mit Computernamen. Die Namen müssen untereinander stehen
   $Computerlist
)

$Computername = $Computerlist + $Computername

Invoke-Command -Computername $Computername -Skriptblock {
  $Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM

  $Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile   -executionpolicy bypass -file $filename

  Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -
User "System" -Deskription "Restart Computer"

  }
```

Achten Sie bei der Addition der Variablen auf die Reihenfolge. Wenn zwei Variablen addiert werden, versucht Powershell den Variablentyp automatisch zu bestimmen. Die Liste aus der Datei wird normalerweise aus mehreren Namen bestehen und damit automatisch von Powershell einem Array zugewiesen. Wenn Sie für den Parameter *–Computername* nur einen Computer eingeben, bekommt die Variable den Typ String (einfacher Text), und Powershell wandelt das Array ebenfalls in Text um, so dass der Inhalt von $Computername dann ungefähr so aussehen könnte:

```
Server1 Server2Server3Server4
```

Das ist dann ein einziger Wert mit einem sinnlosen Computernamen. Um auf Nummer sicher zu gehen, können Sie den Datentyp für Computername auch einfach gleich als Array festlegen, statt Powershell die Zuweisung zu überlassen:

```
Param(
   # Der Pfad der Skriptdatei, die automatisch gestartet werden soll
   $Filename = \\FS1\Skript\RestartComputer.ps1

   # Der Name, der für den geplanten Task vergeben werden soll
   $Taskname = "Restart-Computer",

   # Der Name der Computer, auf denen ein Task angelegt werden soll
   [Array]$Computername,

   # Eine Textdatei mit Computernamen. Die Namen müssen untereinander stehen
   [Array]$Computerlist
)

$Computername = $Computerlist + $Computername

Invoke-Command -Computername $Computername -Skriptblock {
  $Trigger = New-ScheduledTaskTrigger -Daily -DaysInterval 1 -At 1AM

  $Action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument
"-noprofile   -executionpolicy bypass -file $filename

  Register-ScheduledTask -TaskName $Taskname -Action $Action -Trigger $Trigger -
User "System" -Deskription "Restart Computer"
  }
```

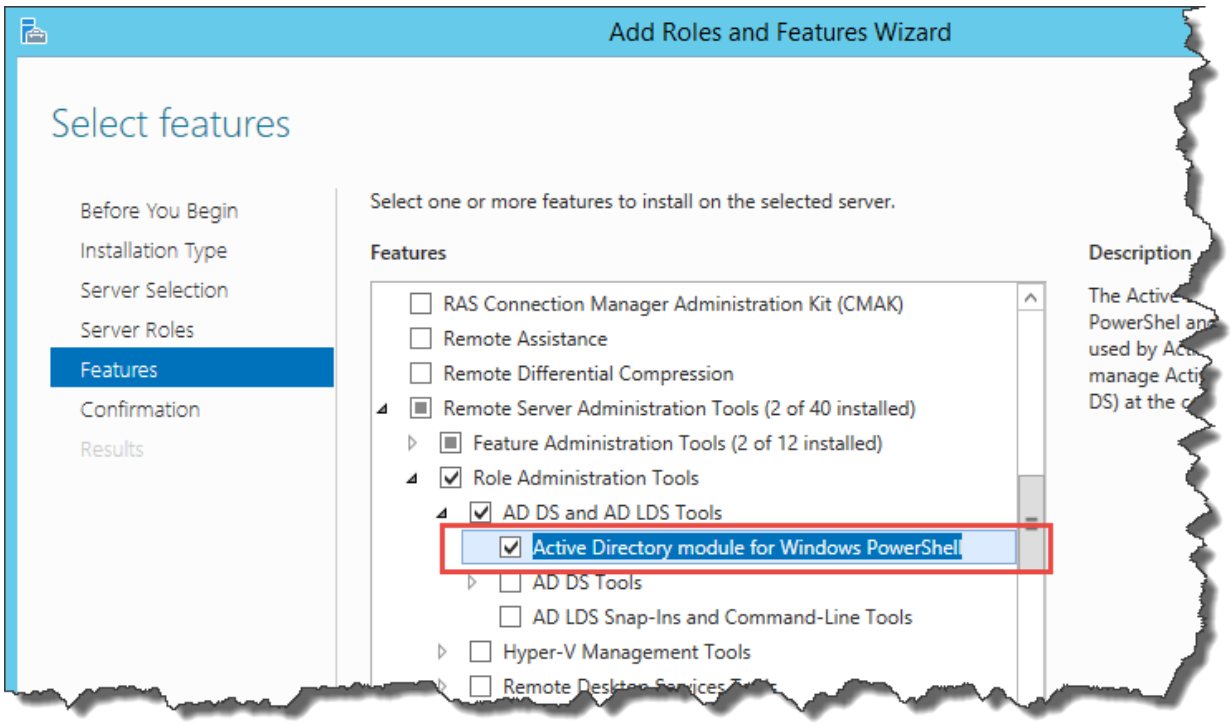
Nun ist es egal, in welcher Reihenfolge Sie die Variablen addieren, Sie erhalten auf jeden Fall immer ein Array.

AD verwalten mit Powershell

Für die Verwaltung des AD stellt Microsoft seit Windows Server 2008 R2 eine Powershell-Erweiterung zur Verfügung. Das Powershell-Modul kann auf jedem Rechner installiert werden und benötigt netzwerkseitig mindestens einen Domänencontroller mit Windows Server 2008 R2, da die Powershell-Cmdlets auf den Active Directory Webservice zurückgreifen, der seit Windows Server 2008 R2 mit allen Domänencontrollern installiert wird. Haben Sie noch keinen Domänencontroller mit Windows Server 2008 R2 in Ihrer Domäne, bietet sich die Active Roles Management Shell von Dell an. Hierunter verbirgt sich eine Powershell-Erweiterung, die ebenfalls alle wichtigen AD-Funktionen zur Verfügung stellt. Sie finden die Management-Shell unter folgendem Link: http://software.dell.com/trials/#!bybrandquestsoftware

Installation

Das Active-Directory Modul und alle Abhängigkeiten werden über den Windows-Server Manager installiert. Starten Sie hierzu den Windows Server Manager, und wählen Sie "Add Roles and Features". Unter Features wählen Sie "Remote Server Administration Tools" -> "Role Administration Tools" -> "AD DS and AD LDS Tools" -> "Active Directory module for Windows Powershell". Für die Verwaltung von Gruppenrichtlinien müssen Sie außerdem das Feature "Group Policy Management" hinzufügen.



Alternativ können Sie die Features auch über die Powershell selbst hinzufügen. Starten Sie hierfür eine Powershell-Konsole mit administrativen Rechten und geben Sie folgenden Befehl ein:

```
Add-Windowsfeature -Name RSAT-AD-PowerShell,GPMC
```

Wenn Sie das Powershell-Modul unter Windows 7 oder Windows 8 / 8.1 nutzen möchten, müssen Sie die Remote Server administration Tools (RSAT) für ihr jeweiliges Betriebssystem herunterladen und installieren. Die RSAT-Tools für Windows 8.1 finden Sie unter folgendem Link:

http://www.microsoft.com/de-de/download/details.aspx?id=39296

Wenn Sie Windows 7 oder Windows Server 2008 R2 mit Powershell 2.0 nutzen, müssen Sie die Module noch importieren – ab Powershell 3.0 findet der Import des Moduls automatisch statt. Der Befehl zum Importieren lautet:

```
Import-Module –name ActiveDirectory
Import-Module –name GroupPolicy
```

Das Powershell-Modul bietet Ihnen, je nach Version (Windows Server 2008 R2, Windows Server 2012) eine ganze Reihe von Cmdlets zur Verwaltung Ihres Active Directory. Um alle Cmdlets aufzulisten, die Ihnen zur Verfügung stehen, nutzen Sie den Befehl

```
Get-command –module ActiveDirectory
Get-Command –module GroupPolicy
```

Die wichtigsten Cmdlets und eine Reihen von Einsatzmöglichkeiten lernen Sie im Folgenden kennen.

Grundlegendes zu den Cmdlets

Bei den AD-Cmdlets handelt es sich um klassische Powershell-Kommandos, die die Standard-Parameter wie *-debug*, *-verbose*, *-whatif*, *-Erroraction* usw. unterstützen. Außerdem ist die Nutzung der Powershell-Pipeline in vielen Fällen sehr nützlich. Die Pipeline ist eine Feature der Powershell, bei der die Ausgabe eines Cmdlets an das jeweils folgende Cmdlet weitergegeben wird. Um die Pipeline zu nutzen, wird das |-Zeichen (senkrechter Strich) verwendet. Mit der Pipeline sind Sie in der Lage, Massenbearbeitungen durchzuführen. In Skripten haben Sie außerdem die Möglichkeit, eine große Anzahl von Objekten mit Foreach-Schleifen zu bearbeiten. Achten Sie darauf, dass die Foreach-Schleife sehr schnell ist, aber alle Objekte in den Speicher lädt und daher sehr Speicherhungrig ist. Die Pipeline ist im Gegensatz dazu deutlich Resourcenschonender, da jedes Objekt einzeln bearbeitet wird, aber auch deutlich langsamer.

Alle Active-Directory Cmdlets tragen AD im Namen. Um z.B. alle Benutzer einer Domäne auszugeben, verwenden Sie das Cmdlet *Get-ADUser*. Um einen AD-Benutzer zu bearbeiten, verwenden Sie *Set-ADUser*. Alle Gruppenrichtlinien-Cmdlets tragen gp im Namen, z.B. *Get-GPPermission*.

Filter

Wenn Sie Objekt aus dem AD abrufen, haben Sie die Möglichkeit, die Objekte zu filtern. Neben den Standard-Varianten der Pipeline (*Where-Object*) können (und müssen) Sie beim Abruf aber einen Filter setzten, der festlegt, welche Objekte zurückgeliefert werden. Die AD-Cmdlets geben Ihnen dafür zwei Möglichkeiten:

Filtern mit dem –Filter-Parameter

Der Filter-Parameter erlaubt Ihnen, die Powershell-Syntax zur Filterung zu verwenden. Möchten Sie beispielsweise alle Benutzer Ihrer Domäne auflisten, setzen Sie als Filter einfach den *:

```
Get-ADUser –Filter *
```

Möchten Sie alle Benutzer auflisten, die mit 'S' anfangen, nutzen Sie die gleiche Abfragesyntax wie beim *Where-Object*:

```
Get-ADUser –Filter { name –like 'S*' }
```

Bei komplexen Filtern setzten Sie immer einen Skriptblock { … } um die Abfrage. Die Vergleichsoperatoren bei Powershell werden nicht wie mathematische Operatoren abgekürzt, sondern über einen "Sprachlichen" Operator. Die Wichtigsten sind:

Operator	entspricht
-eq, -ne	Equal (=), gleich, Not Equal (<>), ungleich
	Lesser then (<), kleiner als bzw. lesser or equal (<=), kleiner gleich

-lt, -le	
-gt , -ge	Greater then (>), größer als bzw. greater or equal (>=), größer gleich
-like, -notlike	wie, nicht wie: Vergleich mit Wildcards (*,[ABC]…)
-in, -contains	Vergleich mit einer Auflistung (name –in ('Alfred','Hans','Harry'); (('Alfred','Hans','Harry' –contains name);
-match	Vergleich mit Regular Expressions

Mehrere Vergleiche können über *-or* bzw. *-and* verknüpft werden. Eine komplette Auflistung aller Operatoren bekommen Sie, wenn Sie in der Powershell eingeben:

```
Get-help about_comparison_operators
```

Filtern mit dem LDAP-Filter

Alternativ können Sie auch einen LDAP-Filter einsetzen, um die Rückgabe einzuschränken. Verwenden Sie hierfür den Parameter *-LDAPFilter*. LDAPFilter können sehr komplex werden. Eine Einführung in das Erstellen von LDAP-Filtern finden Sie in Anhang.

```
Get-ADUser –LDAPFilter "(samaccountname=SQL3)"
```

Mit dem Parameter *-Searchbase* können Sie festlegen, wo im AD gesucht werden soll. Möchten Sie also z.B. alle Konten auflisten, die sich in der OU Domain Controllers befinden, verwenden Sie folgende Abfrage:

```
Get-ADObject -Filter * -SearchBase 'ou=Domain Controllers,dc=contoso,dc=com'
```

Für den Parameter *-Searchbase* verwenden Sie den DN (Distinguished Name) des Objekt, das Sie durchsuchen möchten.

Benutzerverwaltung

Zur Verwaltung von Windows-Benutzern benötigen Sie die folgenden 4 Cmdlets:

- Get-ADUser
- Set-ADUser
- New-ADuser
- Remove-ADuser

Get-ADuser liefert Ihnen Active-Diretory-Benutzer zurück, *Set-ADUser* führt Änderungen an bestehenden Benutzerobjekten durch, *New-ADUser* legt neue Benutzerkonten an und *Remove-ADUser* löscht Benutzerkonten. Alle Befehle sind Pipelinefähig.

```
# Ausgeben aller Benutzer im AD:
Get-ADuser –Filter *

# Ausgeben aller Benutzer in der OU Benutzer in der OU Hannover:
Get-ADUser –Filter * -Searchbase 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com'

# Für alle Benutzer in der OU Benutzer das Attribut City auf Hannover setzen:
Get-ADUser –Filter * -Searchbase 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com' | Set-
ADUser –city 'Hannover'

# Alle Benutzerkonten in der OU Benutzer aktivieren
Get-ADUser –Filter * -Searchbase 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com' | set-
ADUser –Enabled $true

# Alle Benutzerkonten in der OU Benutzer löschen
Get-ADUser –Filter * -Searchbase 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com' |
Remove-ADUser
```

Anlegen von AD-Benutzern

Zum Anlegen von AD-Benutzern verwenden Sie das Cmdlet *New-ADUser*. Das Cmdlet *New-ADUser* hat für jede AD-Eigenschaft einen eigenen Parameter, den Sie angeben können.

```
New-ADUser -SamAccountName 'Holger' -Name 'Holger' -Surname 'Voges' -GivenName
'Holger'
```

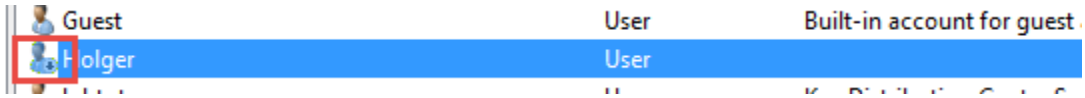


Abbildung 8 - der Pfeil zeigt an, dass der Benutzer deaktiviert ist.

Der neue Benutzer ist nach dem Anlegen deaktiviert - um den Benutzer zu aktivieren, benötigt er ein Kennwort. Da das Kennwort als Secure String übergeben werden muss (ein im Speicher verschlüsseltes Kennwort), muss das Klartextkennwort über das Cmdlet *Convertto-Securestring* erst in ein sicheres Kennwort umgewandelt werden.

```
$SecPwd = ConvertTo-SecureString -String 'Passw0rd' -AsPlainText -Force

New-ADUser -SamAccountName 'Holger' -Name 'Holger' -Surname 'Voges' -GivenName
'Holger' -AccountPassword $SecPwd -Enabled $true
```

Der Parameter *-Enabled* aktiviert den Benutzer, der Parameter *-AccountPassword* legt das Kennwort fest. Hier wird einfach das Kennwort übergeben, das vorher in *$SecPwd* gespeichert wurde.

Um den Benutzer in einer bestimmten OU anzulegen, wird der Parameter Path verwendet:

```
New-ADUser -SamAccountName 'Hans' -Name 'Hans Wurst' -Surname 'Wurst' -GivenName
'Hans' -Path 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com'
```

Wenn Sie Eigenschaften setzten möchten, die Ihnen das Cmdlet *New-ADUser* nicht anbietet, verwenden Sie den Parameter *-OtherAttributes* und übergeben ein Hash-Array. Hash-Arrays sind Namen-Werte Paare. Jedem Wert wird hier ein Name zugewiesen. Hash-Arrays erkennt man an einem @-Symbol, gefolgt von einer geschweiften Klammer:

```
@{'msDS-PhoneticDisplayName'='HolgerVoges'}
```

Der Eintrag *msDS-PhoneticDisplayName* ist der Name des übergebenen Werts, *HolgerVoges* der Wert an sich. Mehrere Namen-Werte-Paare werden mit einem ";" getrennt:

```
@{title="chef";mail="holger.voges@netz-weise.de"}
```

Um also einen Titel und eine E-Mail-adresse beim Anlegen des Benutzers anzugeben, verwenden Sie

```
New-ADUser -SamAccountName 'Hans' -Name 'Hans Wurst' -Surname 'Wurst' -GivenName
'Hans' -Path 'OU=Benutzer,OU=Hannover,DC=Contoso,DC=Com' -OtherAttributes
@{title="Clown";mail="hans.wurst@contoso.com"}
```

Massenanlegen von Benutzerkonten

Die Pipeline macht es Ihnen sehr einfach, eine große Anzahl von Benutzern anzulegen. Am einfachsten geht das über eine CSV-Datei (Comma separated Value), da dieses Format von Excel direkt unterstützt wird. Alles, was Sie benötigen, ist eine Excel-Datei, die in der Überschriftenspalte die Namen der einzelnen Properties enthält:

SamAccountName	Name	Surname	GivenName	City
HansWurst	Hans Wurst	Wurst	Hans	Hannover
KarlKäse	Karl Käse	Käse	Karl	Hannover

Mit dem Standard-Cmdlet *Import-CSV* können Sie die csv-Datei direkt einlesen und per Pipeline an das Cmdlet *Import-CSV* weiterleiten:

```
Import-Csv –path c:\Daten\user.csv –delimiter ';' –Encoding 'Unicode' | New-ADUser
```

Achten Sie aber darauf, dass der Import nur dann funktioniert, wenn die Spaltennamen exakt so heißen wie die Benutzer-Attribute. Außerdem klappt der Import ohne Angabe der Attribute erst mit Windows Server 2012. Wenn Sie noch ältere Versionen der AD-Cmdlets einsetzen, müssen Sie den Befehl *New-ADUser* noch ein einen *Foreach* einschließen und die Attributszuweisungen von Hand vornehmen.

```
Import-Csv –path c:\Daten\user.csv –delimiter ';' –Encoding 'Unicode' |
Foreach-Object { New-ADUser –SamAccountname $_.SamAccountname –Name
$_.Name –Surname $_.Surname –Givenname $_.Givenname –City $_.City }
```

Der Parameter *-Unicode* des Cmdlets *Import-Csv* ist optional und sorgt dafür, dass die CSV-Datei als Unicode eingelesen wird, damit die Umlaute korrekt dargestellt werden. Das setzt allerdings voraus, dass auch die Datei als Unicode gespeichert wurde. Falls Sie mit der Darstellung Probleme bekommen, spielen Sie einfach mal ein wenig mit den anderen Parameter für das Encoding herum.

Mit dem kleinen Tool CSV2AD wird das importieren über eine grafische Oberfläche noch einfacher. Es ist kostenlos und kann bei NTguys herunter geladen werden: http://nt-guys.com/csv2ad-active-directory-user-import-aus-csv/

Ausgeben von AD-Konten

Für die Ausgabe von Benutzern kann das Cmdlet *Get-ADUser* vewendet. Mit Hilfe von *Export-CSV* kann man die Benutzerkonten auch sehr einfach exportieren:

```
Get-ADuser –Filter * | Export-CSV –Path c:\Daten\User.csv –Delimiter
';' -NoTypeInformation
```

Die AD-Cmdlets geben standardmäßig nur ein Subset von Eigenschaften zurück, um die Menge an Daten zu begrenzen, die übertragen und gespeichert werden müssen. Um (fast) alle Eigenschaften eines Objekts zu erhalten, verwenden Sie den Parameter Properties:

```
Get-ADUser –Filter * -Properties *
```

Sie können aber auch einzelne Eigenschaften angeben, was im Zweifelsfall immer schneller geht und resourcenschonender ist. Geben Sie hierfür einfach statt des * die Properties kommasepariert an, die sie benötigen:

```
Get-ADUser –Filter * -Properties City,Manager
```

Wenn Sie Ihr AD nach gesperrten, deaktivierte oder lange nicht mehr genutzten Benutzerkonten absuchen möchten, stellt Ihnen Powershell ein eigenes Cmdlet bereit: *Search-ADAccount*. *Search-ADAccount* hat eine ganze Reihe von Parametern, die filtern, welche Objekte angezeigt werden sollen:

Parameter	Funktion
AccountDisabled	Zeigt alle deaktivierten Konten an
AccountExpired	Zeigt alle Konten an, deren Ablaufzeit überschritten ist. Diese Konten werden automatisch deaktiviert
AccountExpiring	Konten, die bald ablaufen werden

AccountInactive	Konten, die seit einer bestimmten Zeitspanne keine Anmeldung mehr durchgeführt haben
LockedOut	Gesperrte Konten (durch fehlerhafte Anmeldung)
PasswordExpired	Konten, deren Kennwort abgelaufen ist
PasswordNeveExpires	Konten, die die Eigenschaft "Kennwort läuft nie ab" gesetzt haben

Zusätzlich stehen die Parameter *-ComputersOnly* und *-UsersOnly* zur Verfügung, um die Suche auf eine dieser beiden Kontentypen einzuschränken. Mit den Parametern *-Timespan* und *-Datetime* kann eine Zeitspanne oder ein Datum festgelegt werden, auf das gefiltert wird. Um z.B. alle ausgesperrten Benutzerkonten zu suchen, die gesperrt sind, kann man dieses Kommando verwenden:

```
Search-ADaccount –Lockedout –UsersOnly
```

Um alle Konten direkt wieder zu entsperren, leitet man das Ergebnis an *Unlock-ADAccount*:

```
Search-ADaccount –Lockedout –UsersOnly | Unlock-ADAccount
```

Um alle Benutzerkonten anzuzeigen, die sich seit 30 Tagen nicht mehr angemeldet haben, verwendet man:

```
Search-ADaccount –Inactive –Timespan 30 –UsersOnly
```

Der Zeitwert kann bis zu Bruchteilen von Sekunden genau angegeben werden. Es wird folgendes Format verwendet:

D.H:M:S.F

D steht für Tage, H für Stunden, M für Minuten, S für Sekunden und F für Bruchteile von Sekunden (Fractions)

Um z.B. alle Accounts anzuzeigen, die in den nächsten 10 Tagen, 5 Stunden und 30 Minuten ablaufen, gibt man an:

```
Search-ADAccount –AccountExpiring –Timespan 10.5:30
```

Um alle Accounts anzuzeigen, die am 24.12.2020 ablaufen, nutzt man den Parameter Date:

```
$datum = get-date -Year 2020 -Month 12 -Day 24

Search-ADAccount –AccountExpiring –Date $datum
```

Ausgabebegrenzung

Active Directory begrenzt die maximale Anzahl von Objekten, die pro Anfrage zurückgegeben werden, auf 1000. Diese Begrenzung soll den Domänencontroller schützen. Die Anzahl der zurückgegebenen Werte kann aber durch den Parameter *-ResultsetSize* verändert werden. Standardmäßig hat *-Resultsetsize* einen Default-Wert von *$Null*, was die Ausgabe auf unbegrenzt stellt. Man kann die Menge der zurückgelieferten Werte aber einschränken, indem man einen anderen Wert für *-Resultsetsize* setzt:

```
Get-ADUser –Filter * -ResultsetSize 10000
```

Es gibt aber eine weitere Grenze, die durch den die LDAP-Abfrage-Richtlinien gesetzt wird. Und zwar liefert das AD eine maximale Datenmenge von 256 KB zurück.

Organizational Units verwalten

Um Organizational Units im AD anzulegen, stellt Powershell das Cmdlet *New-ADOrganizationalUnit* zur Verfügung. Die Verwendung ist sehr simpel:

```
New-ADOrganizationalUnit –Name Hannover –Path 'DC=Contoso,DC=Com' –Deskription
'Standort Hannover'
```

Das Kommando legt eine neue OU Hannover in der Domäne an, Deskription legt die Beschreibung fest. Das Cmdlet liefert standardmäßig kein Objekt zurück. Will man das Objekt weiterverarbeiten, verwendet man den Parameter *-Passthru*:

```
$OU = New-ADOrganizationalUnit -Name Gruppen -Path 'ou=Hannover,dc=contoso,dc=com'
–PassThru
```

Um sich Organizational Units ausgeben zu lassen, verwendet man *Get-ADOrganizationalUnit*. Auch hier kann wieder der Filter-Parameter verwendet werden:

```
Get-ADOrganizationalUnit Filter { name –eq 'Hannover' }
```

Gruppen und Gruppenmitgliedschaften

Eine Reihe von AD-Cmdlets machen das Verwalten von Gruppen einfacher. Um eine neue Gruppe anzulegen, verwenden Sie einfach das Cmdlet *New-ADUserGroup*:

```
New-ADGroup -GroupCategory Security -DisplayName RDPUser -GroupScope DomainLocal
-Name RDPUser -Path 'ou=Gruppen,ou=Hannover,dc=contoso,dc=com' -SamAccountName
RDPUser
```

Mit *-GroupScope* legen Sie den Typ der Gruppe fest (Domänenlokal, Global, Universal), mit *-Groupcategory* legen Sie fest, ob es sich um eine (email-)Verteilergruppe handelt, und *-SamAccountname* legt den Gruppenamen fest.

Um einer Gruppe Mitglieder hinzuzufügen, verwenden Sie *Add-ADGroupmember*:

```
Add-ADGroupMember -Identity RDPUser -Members Holger
```

Dank der Pipeline ist es sehr einfach, einer Gruppe eine große Anzahl von Benutzern in einem Rutsch hinzuzufügen. Nutzen Sie *Get-ADUser* zum Ausgeben der Benutzer und pipen Sie das Ergebnis einfach in *Add-ADGroupMember*. Da *Add-ADGroupmember* nicht von Haus aus Pipelinefähig ist, nutzen wir das Cmdlet *Foreach-Object*:

```
Get-ADUser –Filter { name –like 'RDP* } | Foreach-Object { Add-ADGroupmember –
Identity RDPUser –Members $_ }
```

Um die Gruppe mit Ihren Mitgliedern wieder auszugeben, verwenden Sie *Get-ADGroupmember*:

```
Get-ADGroupMember –Identity 'RDPUser'
```

Get-ADGroupMember kann auch rekursiv für alle Gruppen, die in der Gruppe Mitglied sind, wieder die Gruppenmitgliedschaft auflösen, so dass am Ende nur noch die Benutzerobjekte aufgelistet sind:

```
Get-ADGroupMember –Identity 'RDPUser' -Recursive
```

Wenn Sie nur die Gruppeninformationen benötigen, verwenden Sie *Get-ADGroup*:

```
Get-ADGroup –Identity RDPUser
```

Um alle Gruppen anzuzeigen, in denen ein Benutzer sich befindet, überprüfen Sie am besten das Benutzer-Attribut *MemberOf*:

```
Get-AdUser -Filter { name -eq 'Holger' } -Properties MemberOf |
select-object -Property MemberOf
```

Sie können sich die Gruppe auch direkt holen, indem Sie die Gruppe(n) an *Get-ADGroup* weiter pipen:

```
Get-AdUser -Filter { name -eq 'Holger' } -Properties MemberOf |
select-object -Property MemberOf | get-ADGroup
```

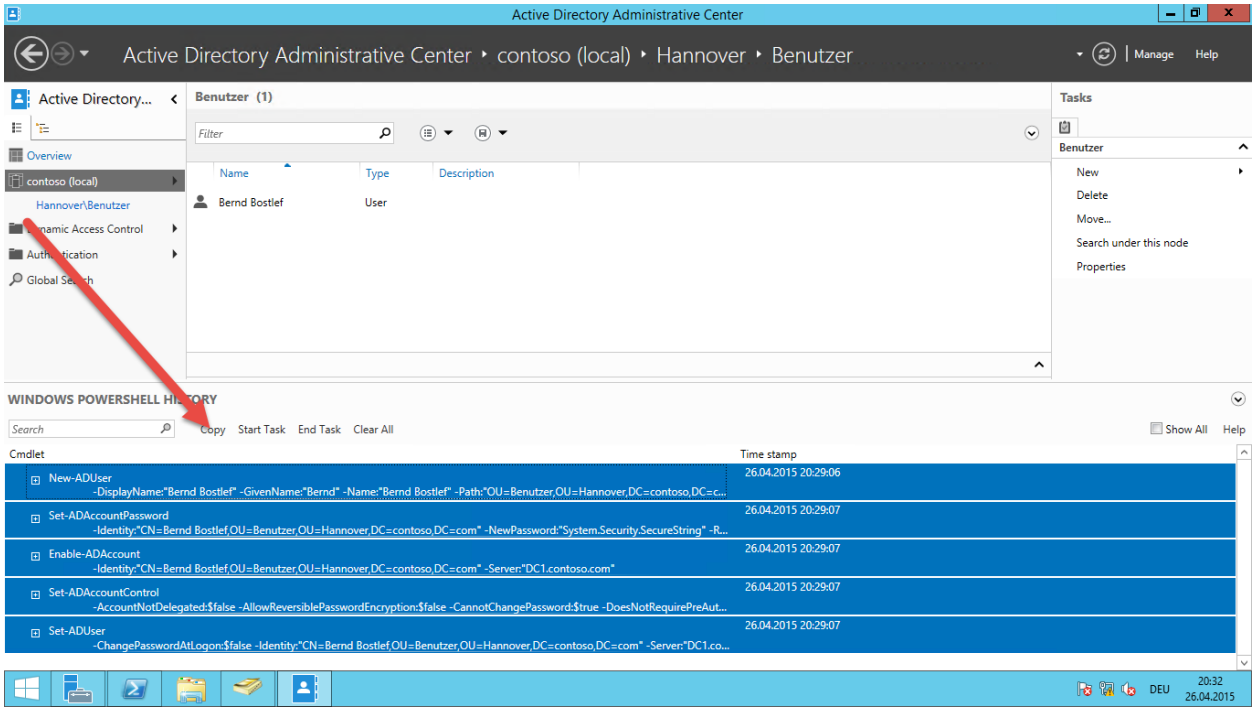
Beliebige AD-Objekte verwalten

Um beliebige AD-Objekte aufzurufen, verwenden Sie das Cmdlet *Get-ADObject*. Die Syntax ist ähnlich wie bei *Get-ADUser*:

```
Get-ADObject -Filter * -SearchBase 'ou=Hannover,dc=contoso,dc=com'
```

So erleichtern Sie sich das Erstellen von Powershell-Skripten

Mit Powershell können Sie sämtliche wiederkehrenden Aufgaben im AD automatisieren. Wenn Sie mal nicht wissen, welchen Befehl Sie anwenden sollen, hilft Ihnen das Active Directory Administrative Center ab Windows Server 2012 weiter. Dieses Programm basiert komplett auf Powershell und zeigt alles, was Sie hier durchführen, auch noch mal als Powershell-Befehl an. Starten Sie hierfür einmal das Administrative Center und legen Sie einen neuen Benutzer an. Wenn Sie nun im Administrative Center auf die unterste Zeile schauen, finden Sie hier die Windows Powershell History. Klicken Sie auf den kleinen Pfeil ganz rechts des Textes *Windows Powershell History*, so öffnet sich die Liste der zuletzt ausgeführten Powershell-Befehle. Sie brauchen jetzt nur noch die Befehle zu markieren, die Sie sich anzeigen lassen wollen, und können diese über Copy ins Clipboard kopieren.



Anhang A

LDAP-Filter verstehen und erstellen

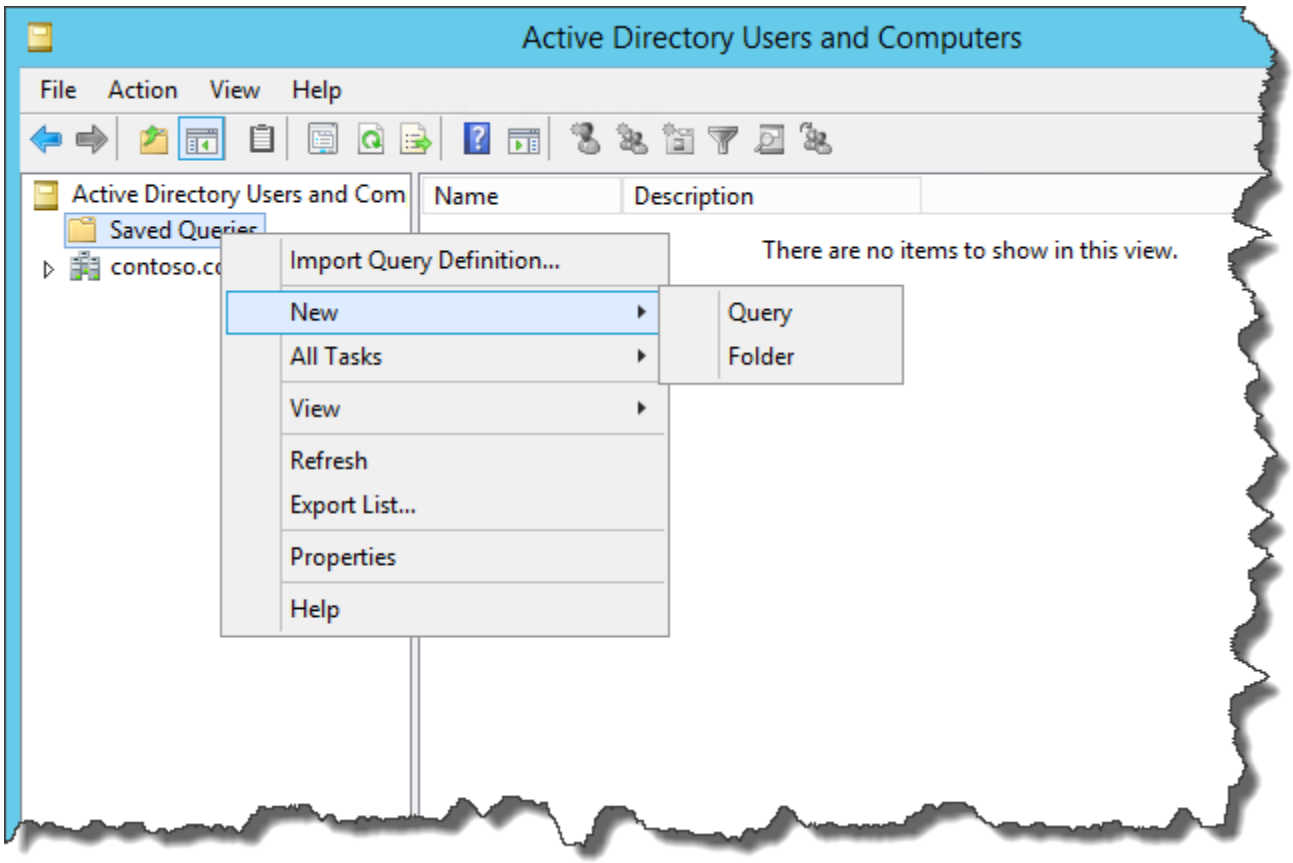
Was ist LDAP überhaupt?

Active Directory ist eine Datenbank, die dem LDAP-Standard entspricht. LDAP bedeutet Lightweight Directory Access Protocol und ist eine Untermenge von X.500, einem Standard zum Speichern von Personendaten. X.500 ist unter anderem ein Verfahren, das angetreten war, um emails weltweit zu versenden, aber auch um Benutzerinformationen zu speichern. X.500 stellte sich aber letztendlich als zu komplex heraus, und wurde schnell durch SMTP überholt und abgelöst. Verzeichnisdienste (also Datenbanken mit z.B. Benutzerinformationen) verwenden seit langer Zeit eine Untermenge von X.500, die deutlich einfacher ist: LDAP.

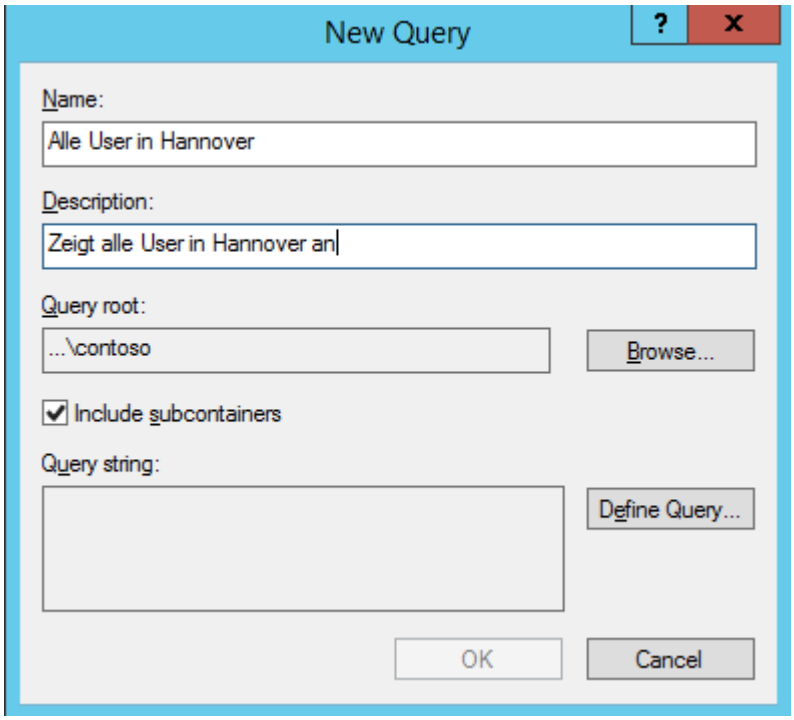
LDAP-Datenbanken sind hierarchisch aufgebaut, also in Baumstrukturen. Um Daten aus der Datenbank zu filtern, verwendet man sogenannte LDAP-Abfragen. Jeder LDAP-Browser, wie z.B. das Active-Directory Benutzer und Computer, setzen LDAP-Abfragen ein, um die AD-Datenbank abzufragen.

Einfaches erstellen einer LDAP-Abfrage

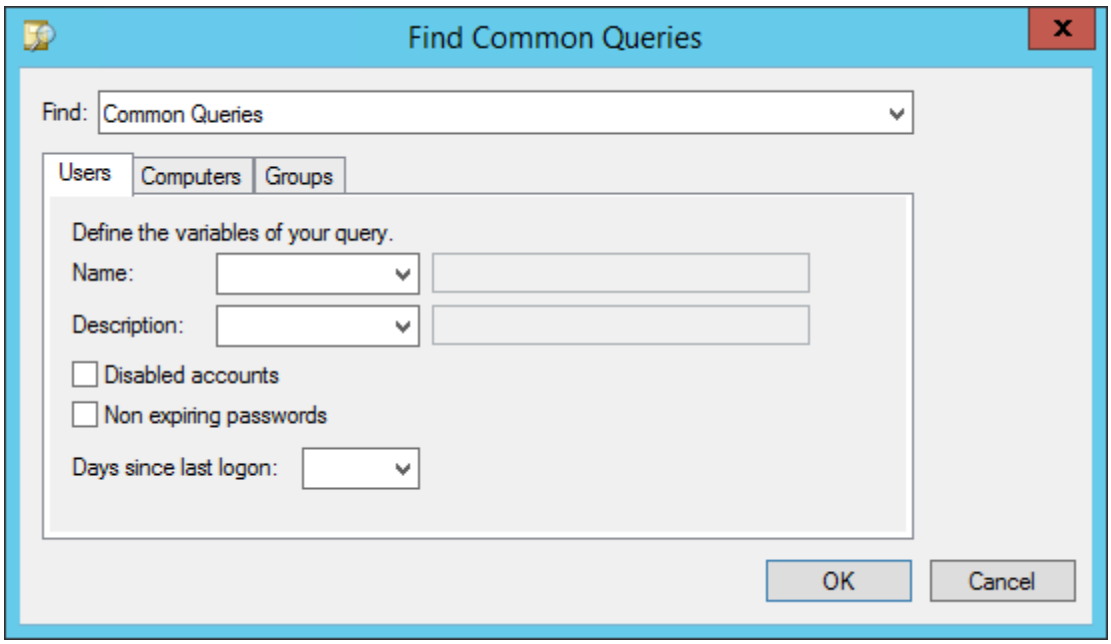
Wenn Sie mit Active Directory arbeiten, haben Sie ein sehr nützliches grafisches Hilfsmittel zur Hand, um Abfragen zu erstellen: Active Directory Benutzer und Computer. Wenn Sie Active Directory Benutzer und Computer starten, finden Sie links oben im Baum einen Eintrag "gespeicherte Abfragen" oder "Saved Queries". Diesen Eintrag können Sie starten, um eine neue Abfrage zu erstellen:



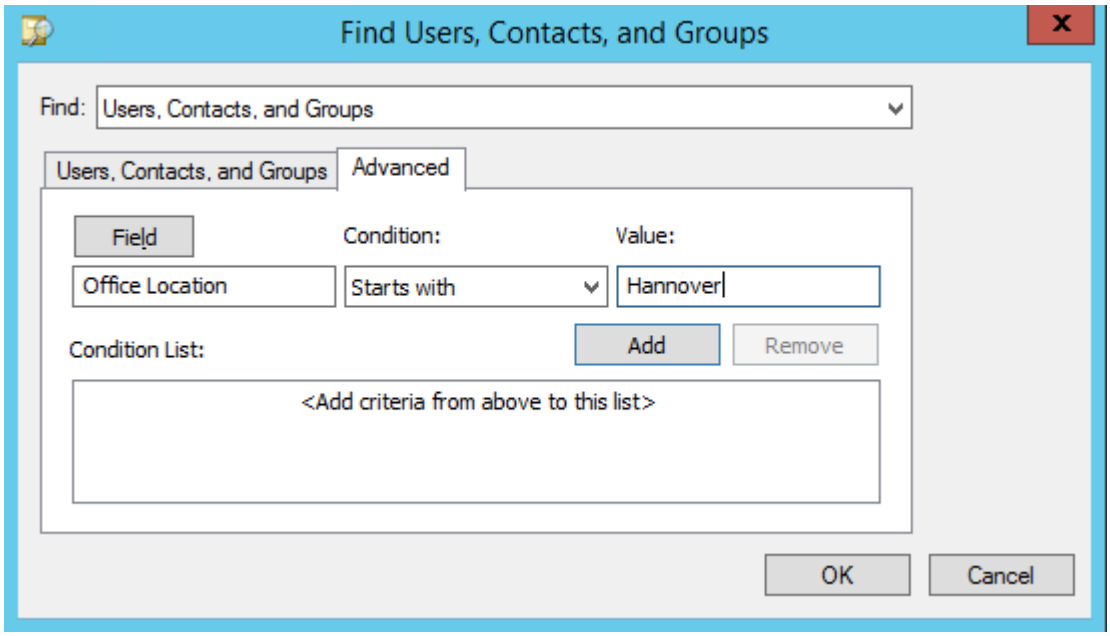
Öffnen Sie den Abfrage-Editor unter "gespeicherte Abfragen"



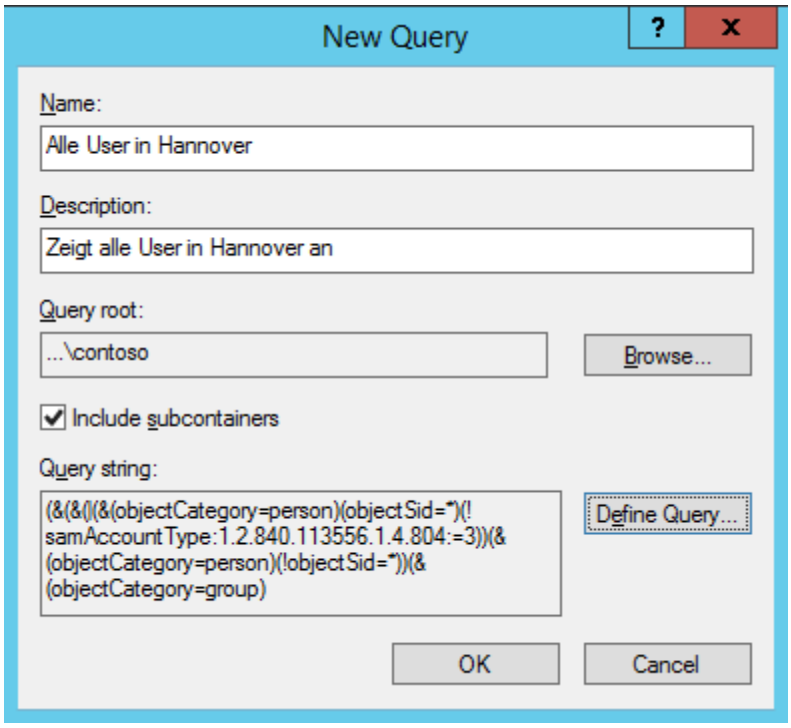
Legen Sie den Namen der Abfrage fest



Wechseln Sie von Common Queries zu Users, Contacs, Groups



Im Reiter Advanced legen Sie die Benutzerattribute fest, die sie abfragen wollen



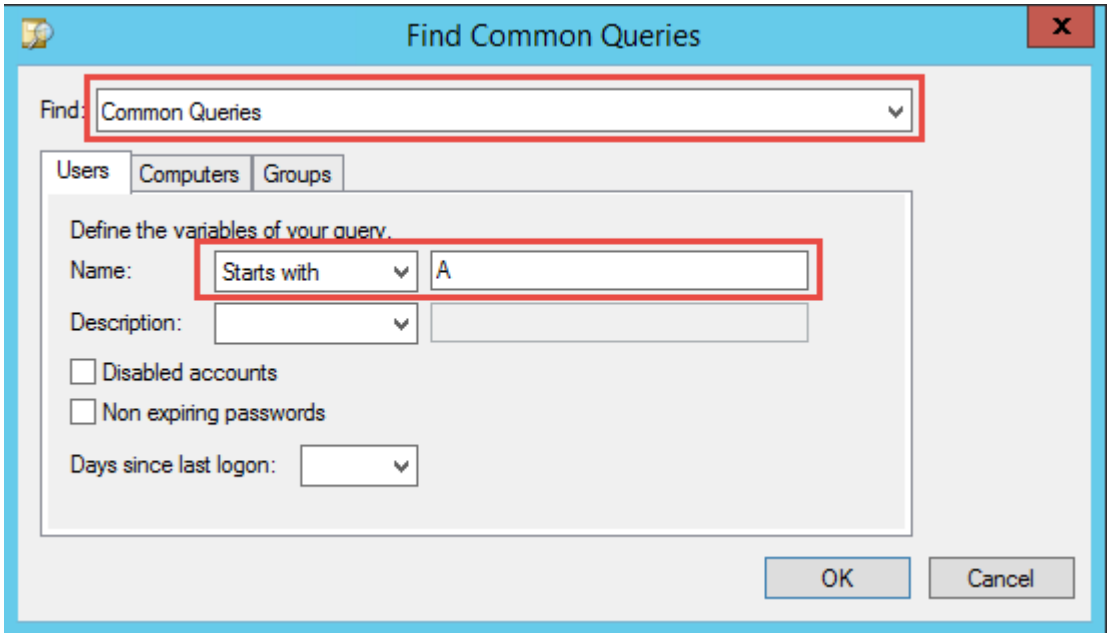
Es wird eine LDAP-Abfrage erzeugt

Verstehen und bearbeiten der Abfrage

Bei einem LDAP-Abfrage werden zusammenhängende Teilabfragen in Klammern gestellt: *(...)* wäre eine zusammenhängende Teilabfrage. Um mehrer Abfragen miteinander zu verknüpfen, werden die zu verknüfenden Abfragen wiederum in eine äußere Klammer gestellt: *((...) (...) (...))*. Wie die inneren Abfragen verknüpft werden, wir durch einen Operator festgelegt, der nach der ersten geöffneten Klammer gestetzt wird. Es gibt drei wichtige Operatoren:

! = NOT
& = AND
| = OR

Eine Abfrage, bei der alle drei in der Klammern zusammenhängenden Abfragen UND-Verknüpft werden, sähe also so aus: (& (...) (...) (...)). Es werden also nur die Suchergebnisse ausgegeben, die alle 3 Bedingungen erfüllen.
Das unangenehme ist, dass die Abfrage, die von AD Benutzer und Computer erzeugt werden, so fürchterlich komplex aussehen. Deshalb nehmen wir erst einmal eine ganz einfache Abfrage:



Man achte darauf, dass unter Suchen „Allgemeine Abfrage“ eingetragen ist, und der Reiter „Benutzer“ ausgewählt ist.

Die Abfrage, die erzeugt wird, sieht so aus:

(&(objectCategory=user)(userPrincipalName=A*))

Die ersten Klammer legt fest, dass nur Benutzerobjekte ausgegeben werden sollen, die zweite Klammer legt fest, dass die Eigentschaft UserPrincipalName mit A beginnen soll und danach beliebige Zeichen erlaubt sind (Wildcard: *).

Eine Abfrage kann allerdings auch kompliziertere Formen annehmen, wie wir in unsere obigen Abfragen gesehen haben:

(&(&(|(&(objectCategory=person)(objectSid=*)(!samAccountType:1.2.840.113556.1.4.804:=3))(&(objectCategory=person)(!objectSid=*))(&(objectCategory=group)(groupType:1.2.840.113556.1.4.804:=14)))(objectCategory=user)(objectClass=user)(physicalDeliveryOfficeName=Hannover*)))

Alles klar? Nein? Dann dröseln wir die Anfrage mal auf:

(&(&(|(&(objectCategory=person)(objectSid=*)(!samAccountType:1.2.840.113556.1.4.804:=3))(&(objectCategory=person)(!objectSid=*))(&(objectCategory=group)(groupType:1.2.840.113556.1.4.804:=14))(objectCategory=user)(objectClass=user)(physicalDeliveryOfficeName=Hannover*)))

Die jeweils farbig markierten Abfragen sind die Unterabfragen, die mit einem UND verknüpft sind, was man an dem führenden (nicht markierten) & sind.

Die gelbe Abfragen legt fest, dass wir ein Benutzerobjekt suchen, das den Samaccounttype hat, der hier angegeben ist. Eine Auflistung der einzelnen Werte finden Sie in der MSDN.

(&(objectCategory=person)(objectSid=*)(!samAccountType:1.2.840.113556.1.4.804:=3))

Der grüne Teil der Abfrage legt fest, dass wir ein Objekt vom Typ Person suchen. Person ist die Klasse, aus der alle Userobjekte abgeleitet sind (auch Computer). Das Ausrufungszeichen vor der ObjectSID ist ein NOT und negiert die Aussage, die folgt. Es ist also keine beliebige SID erlaubt. Die Objekte, die vom Typ Person sind, aber keine SID haben, sind Kontakte.

(&(objectCategory=person)(!objectSid=*))

Der blaue Teil schließt auch Gruppen in die Abfrage mit ein:

(&(objectCategory=group)(groupType:1.2.840.113556.1.4.804:=14))

Wichtig an dieser Stelle ist das | direkt vor dem gelben Block. Der senkrechte Strich ist ein OR und legt fest, dass entweder der gelbe ODER der grüne Block oder der blaue Block zutreffen müssen. Ursache für diese umständliche Suche ist der Query-Editor, denn wir haben bei der Abfrage ja nach "Benutzer, Kontakte und Gruppen" gesucht.

Der graue Block ist UND-Verknüft (die & am Anfang der Abfrage) und legt den zweiten Teil unserer Bedingung fest - dass das Feld OfficeLocation mit Hannover starten soll.

Prinzipiell sehen Sie also, dass eine LDAP-Abfrage gar nicht so schwer ist, wenn Sie die Objekte kennen, nach denen Sie suchen, und den Überblick über die Klammern nicht verlieren. Eine sehr hilfreiche Website, wenn Sie tiefer in das Thema LDAP einsteigen möchte, ist http://www.SELFADSI.de.

Über den Autor

Holger Voges ist IT-Trainer und Consultant. Seine IT-Karriere begann mit einem Atari ST 512 Mitte der 80er Jahre. Seine ersten Erfahrungen mit großen Netzwerken hat er im Systembetrieb der Volkswagen Financial Services 1999 gewonnen. Ab dem Jahr 2000 war er dann als freiberuflicher IT-Trainer für verschiedene Schulungsunternehmen im Bereich Braunschweig und Hannover tätig, bevor er 2002 mit 2 Mitstreitern sein erstes Schulungsunternehmen LayerDrei in Braunschweig gegründet hat. Nach seinem Ausstieg bei LayerDrei war er dann mehrere Jahre als freiberuflicher Consultant vor allem im SQL-Server Umfeld u.a. für T-Home Entertain, e.on und Hewlett-Packard unterwegs, bevor er 2012 das Schulungsunternehmen Netz-Weise gegründet hat.

Netz-Weise hat sich auf Firmenschulungen im profesionellen IT-Umfeld spezialisiert und bietet Schulungen u.a. im Bereich Microsoft, VMWare, Linux und Oracle an.