Programmieren mit der Windows PowerShell

von

Nils Gutsche

(aka Vellas)

Einleitung

Im November 2006 erschien die finale Fassung der Windows PowerShell. Sie gilt als der Nachfolger der Eingabeaufforderung (cmd.exe) und soll zudem auch den Windows Script Host ablösen. Bisher konnte man die Möglichkeiten der Windows-Eingabeaufforderung eigentlich nur belächeln. Gegenüber einer Unix-Shell waren die Möglichkeiten in Windows bisher ein Witz. Dies soll sich mit der Windows PowerShell ändern. Die PowerShell wurde mit .NET entwickelt und setzt auch auf dem .NET-Framework auf. Der Anwender kann mit PowerShell-Skripts die ganze Stärke des .NET-Frameworks nutzen. Daher sind Kenntnisse in .NET von Vorteil bzw. sollten sich angeeignet werden. Auch Variablen entsprechen den Datentypen aus dem .NET-Framework, wodurch so auf die Methoden und Eigenschaften der Objekte zugegriffen werden kann.

Dieses Tutorial soll einen Einstieg in die Grundlagen der Programmierung mit der PowerShell geben. Am Ende dieses Dokuments gibt es noch einen kleinen Einstieg in Funktionen und Skripts, auf dem sich aufbauen lassen sollte um dann mit Hilfe des .NET-Frameworks eigene kleine Skripts zu schreiben.

Variablen

Variablen, ein wichtiger Bestandteil jeder Programmier-/Skriptsprache, so auch der PowerShell. Eine Variable beginnt vor dem Bezeichner mit einem Dollarzeichen:

```
$var = "Hallo Welt!"
```

Der Variablentyp passt sich automatisch dem Typ des zugewiesenen Inhalts an. Im obigen Beispiel ist `$var` also vom Typ string. Wer sicherstellen möchte, dass eine Variable von einem bestimmten Typ ist und nicht mit anderen Werten initialisiert werden soll, kann dies durch Angabe des Typs vor der Variablen sicherstellen:

```
[int]$zahl = 5
```

So kann sichergestellt werden, dass in der Variablen nur Zahlen vom Typ Integer stehen können. Würde man im Beispiel versuchen `$zahl` einen Wert vom Typ string, dessen Zeichenfolge nicht in einen Integer konvertierbar wäre, zuzuweisen würde eine Fehlermeldung ausgegeben werden:

OK: `[int]$zahl = "5"`

Falsch: `[int]$zahl = "A"`

Die erste Zuweisung ist ok, da `"5"` in einen Integer konvertiert wird. `"A"` kann nicht in einen Integer konvertiert werden, daher erzeugt diese Zuweisung eine Fehlermeldung.

Als Typ können alle aus dem .NET-Framework bekannten Datentypen verwendet werden. Für Zahlen und Zeichen sind das die folgenden:

Zahlen: [int], [long], [double], [decimal], [float], [single], [byte]

Zeichen: [string], [char]

Des Weiteren kann in der PowerShell auch ein Gültigkeitsbereich für die Variablen festgelegt werden. Es gibt folgende Gültigkeitsbereiche:

- Global: Überall sichtbar
- Script: In allen Bereichen der Skriptdatei sichtbar
- Local: Nur im aktuellen Bereich und Unterbereichen sichtbar
- Private: Nur im aktuellen Bereich sichtbar

Wird kein Gültigkeitsbereich angegeben für eine Variable gilt per Default „Local“ als Gültigkeitsbereich. Möchte man z.B. eine Variable haben die überall sichtbar ist geht das wie folgt:

```
$global:z = "A"
```

Variablen werden bei der ersten Verwendung automatisch angelegt und initialisiert. Wer lieber sicherstellt, dass seine Variablen immer korrekt initialisiert sind, kann dies mit der Ausführung des folgenden Befehls sicherstellen:

```
set-psdebug –strict
```

Wird eine Variable jetzt vor der ersten Verwendung nicht initialisiert, wird eine Fehlermeldung ausgegeben.

Auf Methoden und Eigenschaften von Objekten greift man mit Hilfe des Punktoperators zu. Ist man sich mal nicht über den kompletten Namen einer Eigenschaft oder einer Methode im Klaren, so kann man die ersten Buchstaben dieser eintippen und mit Tab den Namen durch die Command-Completion ausfüllen lassen. Außerdem erkennt man immer gleich ob es sich um eine Methode oder eine Eigenschaft handelt:

```
$a="abcdef"

$a.len<Tab>

$a.Length
```

Wenn man sich nicht sicher ist wie eine Methode/Eigenschaft heißt, kann man auch einfach den Punkt hinter der Variablen setzen oder auch noch das ein oder andere Zeichen um die Suche ein wenig einzugrenzen und so durch mehrfaches Drücken der Tabulatortaste durch die Eigenschaften und Methoden springen. Wollen wir die Methoden/Eigenschaften die mit 'S' beginnen durchsuchen bis wir die Funktion „Substring" gefunden machen wir das wie folgt:

```
$a.S<Tab>

$a.Split(<Tab>

$a.StartsWith(<Tab>

$a.Substring(
```

Anschließend können wir der Funktion nach Übergabe der nötigen Parameter aufrufen:

```
$a.Substring(2)
```

Dieser Aufruf von Substring würde alles ab Index 2 ausgeben, also den String „cdef", da in .NET die Indizierung mit 0 beginnt.

Operatoren und Befehle

Die PowerShell unterstützt auch alle arithmetischen Operatoren die man für das einfache Rechnen benötigt, also + (Addition), - (Subtraktion), * (Multiplikation), / (Division) und % (Modulo; Rest einer Division):

```
$a = 1

$a + 2
```

Die o.g. Operatoren gibt es auch mit einem Gleichheitszeichen dahinter. Diese Operatoren rechnen auf den Wert einer Variablen den Wert der rechten Seite drauf und speichern diesen wieder in der Variablen, die auf der linken Seite steht:

```
$a = $a + 1
```

ist äquivalent zu:

```
$a += 1
```

Auch in der PowerShell gilt die Punkt-Vor-Strich-Rechnung, um dies zu kontrollieren kann man auch einfache Klammern benutzen:

```
$a = (2 + 2) * 3
```

Den Plus-Operator gibt es auch für Zeichenketten zur Konkatenation von Zeichenketten:

```
$a = "abc"

$b = $a + "d"

$a += "d"
```

Sehr Hilfreich für die Suche in Texten sind die Suchbefehle über Wildcards (-like) und reguläre Ausdrücke (-match). Diese geben bei Erfolg „True“ und bei Misserfolg „False“ zurück:

```
"abc" –like "a*"

"adc" –match "a[bd]c"
```

Diese beiden Befehle arbeiten case-insensitiv, d.h. sie beachten die Groß- und Kleinschreibung nicht. Für den case-sensitiven Vergleich, also unter Beachtung der Groß- und Kleinschreibung, gibt es die Befehle -clike und -match.

Was in keiner Programmiersprache fehlen darf, sind Vergleichsoperatoren. Auch diese besitzt die PowerShell. Allerdings gibt es in der PowerShell nicht die aus anderen Sprachen bekannten Operatoren >, <, = usw. sondern die folgenden (in Klammern steht die Bedeutung und ein Beispiel für Vergleichsoperatoren aus anderen Sprachen):

- -eq (equals; =, ==)
- -ne (not equals; !=, <>)
- -gt (greater than, >)
- -ge (greater or equal, >=)
- -lt (lower than, <)
- -le (lower or equal, <=)

Hier ein Beispiel für einen Vergleich der „True“ liefert:

```
5 –lt 10
```

Arrays

Arrays können sehr leicht erstellt werden. Dazu wird einer Variablen einfach eine kommaseparierte Liste mit Werten übergeben. Auf diese Werte kann anschließend per Indexoperator zugegriffen werden. Die Indizierung eines Arrays beginnt immer bei 0. Die Anzahl Elemente eines Arrays kann mit dem Attribut „Length“ ermittelt werden:

```
$a=1,2,3

$b="Hallo","du","schöne","Welt"

$a.Length

$b.Length

$x=$a[0]

$y=$b[1]
```

Die ersten beiden Kommandos legen jeweils ein Array an. Mit dem dritten und vierten Kommando wird die Länge der beiden Arrays ermittelt. Dies entspricht für $a dem Wert 3 und für $b dem Wert 4. In Zeile 5 und 6 wird jeweils auf ein Element des Arrays zugegriffen. $a[0] liefert 1 und $b[1] liefert „du“ zurück.

Zum Hinzufügen neuer Elemente kann der +=-Operator genutzt werden:

```
$a+=22,23
```

Diese Anweisung fügt dem Array, am Ende, die Elemente 22 und 23 hinzu. Damit verändert sich auch der Wert von „Length“ für $a auf 5.

Bedingungen und Schleifen

In der PowerShell gibt es zwei Möglichkeiten um Bedingungsabfragen zu realisieren. Zum einen die if-Abfrage und die Auswahlabfrage per switch-Befehl. Die if-Abfrage ist wie folgt aufgebaut:

```
if (bedingung) { … }

elseif (bedingung) { … }

else { … }
```

Es können auch mehrere elseif-Abfragen aufeinander folgen. Hier ein Beispiel für eine if-Abfrage, mit zwei elseif-Abfragen:

```
if ($x –gt 100)
{
  "X ist größer als 100."
}
elseif ($x –gt 50)
{
  "X ist größer als 50."
}
elseif ($x –gt 25)
{
  "X ist größer als 25."
}
else
{
  "X ist ziemlich klein."
}
```

Mit Hilfe der Vergleichsoperatoren werden die entsprechenden Bedingungen geprüft, wonach sich entscheidet ob ein Zweig ausgeführt wird oder nicht. Wird kein Zweig zuvor ausgeführt, wird der else-Zweig ausgeführt, der eine Behandlung für alle anderen Fälle die zuvor nicht zutrafen durchführt.

Ähnlich dazu funktioniert der switch-Befehl. Bei diesem Befehl wird, abhängig von einem Ausdruck, ein bestimmter Codeteil ausgeführt. Für den switch-Befehl gibt es auch verschiedene Parameter um das Verhalten beim Vergleich mit den vorhandenen Marken zu steuern. Die möglichen Parameter sind folgende:

-case : Groß- und Kleinschreibung wird bei Stringvergleichen beachtet.

-wildcard: Das Wildcard-Zeichen (*) kann in den Marken genutzt werden.

-regex: Reguläre Ausdrücke können in den Marken genutzt werden.

Die Parameter -wildcard und -regex können mit dem Parameter -case in Verbindung genutzt werden. Es folgen ein paar Beispiele für die Verwendung von switch und seiner möglichen Parameter:

```
switch  (2) { 1 { "Eins" } 2 { "Zwei" } }

switch -case ('abc') { 'abc' { "Klein" } 'ABC' { "Gross" } }

switch -wildcard ('abc') { a* { "A Stern" } *c { "Stern C" } }

switch -regex ('abc') { ^a { "a*" } 'c$' { "*c" } }
```

Im ersten Beispiel wird „Zwei" ausgegeben. Im zweiten Beispiel wird „Klein" ausgegeben und in dem dritten und vierten Beispiel werden jeweils die Texte beider Marken ausgegeben. Für das dritte Beispiel wird also sowohl „A Stern" als auch „Stern C" ausgegeben. Für das vierte Beispiel wird sowohl „a*" als auch „*c" ausgegeben.

Zur Wiederholung mehrerer Befehle gibt es diverse Schleifenkonstrukte. Es gibt eine while-Schleife, eine do-while-Schleife, eine for-Schleife und eine foreach-Schleife. Die while-Schleife ist eine Schleife mit einer Eintrittsbedingung und wird so lange ausgeführt bis die Bedingung zur Ausführung nicht mehr stimmt. Dabei kann es vorkommen, dass die Schleife nie durchlaufen wird. Anders ist da die do-while-Schleife. Diese wird mindestens einmal durchlaufen und dann so oft wiederholt bis auch hier die Bedingung zur Ausführung der Schleife nicht mehr stimmt. Die for-Schleife ist eine Schleife mit einem Zähler, diese wird solange ausgeführt bis die Zählvariable einen bestimmten Wert erreicht hat. Die foreach-Schleife arbeitet solange, bis alle Elemente in einem Container (Array, Liste, …) abgearbeitet sind.

Es folgt eine Übersicht über die 4 Schleifenkonstrukte:

```
while (bedingung) { … }

do { … } while (bedingung)

for (zäler initialisieren; bedingung; zähler in-/dekrementieren) { …
}

foreach ($variable in list) { … }
```

Es folgen nun ein paar Beispiele zu den jeweiligen Schleifenkontrukten um aufzuzeigen wie diese arbeiten.

while:

```
$i=0
while ($i -lt 3)
{
  echo $i
  $i++
}
```

Bei dieser Schleife werden die Werte 0, 1 und 2 ausgeben.

do-while:

```
$i=0
do
{
  echo $i
  $i++
} while ($i -lt 1)
```

Diese Schleife wird nur genau einmal ausgeführt und gibt 0 aus. Da nach der ersten Ausführung bereits die Bedingung, um die Schleife nochmal auszuführen, nicht mehr stimmt.

for:

```
for ($i=0; $i -lt 5; $i++) { echo $i }
```

Hier wird erst der Zähler initialsiert mit 0. Die Bedingung bricht die Schleife ab, sobald $i den Wert 5 erreicht hat. $i++ erhöht den Zähler nach jedem Schleifendurchlauf. Damit werden von dieser Schleife die Werte 0, 1, ,2 ,3 und 4 ausgeben.

foreach:

```
$array=1,2,3,4,5
foreach ($element in $array)
{
  echo $element
}
```

In diesem Beispiel werden alle Elemente des Arrays ausgegeben. D.h. es wird 1, 2, 3, 4 und 5 ausgegeben.

Um eine Schleife abzubrechen bei einer bestimmten Bedingung oder um einen Schleifendurchlauf zu überspringen bei einer bestimmten Bedingung gibt es „break“ und „continue“. Dazu folgen zwei einfache Beispiele, die die Arbeitsweise dieser beiden Anwendungen verdeutlicht. Dazu wird nochmal die foreach-Schleife als Beispiel genutzt:

```
$array=1,2,3,4,5

foreach ($element in $array)

{

  if ($element -eq 2)

  {

    echo "Zwei"

    continue

  }

  elseif ($element -eq 4)

  {

    break

  }

  echo $element

}
```

Diese Schleife bricht ab, wenn $element den Wert 4 hat. Wenn $element den Wert 2 hat, wird „Zwei“ ausgegeben statt der Zahl und der Rest der Schleife übersprungen. Somit erhalten wir die Ausgabe 1, Zwei und 3.

Notwendige Maßnahmen zur Ausführung von Skripts

Man kann zwar in der PowerShell ohne weitere Einstellungen vorzunehmen programmieren, aber keine Skripts ausführen. In der Regel schreibt man seinen Code aber doch lieber in eine Datei und führt diese anschließend aus. So kann man jederzeit ohne weiteres Fehler beheben und kann jederzeit erneut auf sein Skript zurückgreifen wenn es benötigt wird. Um ein Skript ausführen zu können müssen wir zunächst einige Einstellungen vornehmen. Microsoft bietet verschiedene Möglichkeiten zur Einstellung der Ausführungsrichtlinien an. Diese Richtlinien legen fest welche Skripts ausgeführt werden dürfen. Es gibt folgende Richtlinien:

- Restricted: Keine Ausführung von Skripts möglich
- AllSigned: Nur signierte Skripts werden ausgeführt
- RemoteSigned: Heruntergeladene Skripts werden nur ausgeführt, wenn Signatur vorhanden
- Unrestricted: Jedes Skript ausführen

Ich rate von der letzten Einstellung („Unrestricted“) ab, da durch diese Möglichkeit ohne weiteres Schadcode in das System eingeschleust werden kann. Bei „RemoteSigned“ brauchen Skripts die vom lokalen Computer ausgeführt werden keine Signatur, daher ist diese Methode auch nicht die beste Wahl. Allerdings hat „RemoteSigned“ den Vorteil das, wenn man ein Skript entwickelt, dieses nicht vor jeder Ausführung signiert werden muss. Die sicherste Alternative ist „AllSigned“, da Skripts nur ausgeführt werden, wenn Sie eine Signatur haben und der Computer auf dem das Skript ausgeführt werden soll, dem Herausgeber der Signatur vertraut. Daher ist meine Empfehlung in Bezug auf die Sicherheit die PowerShell auf „AllSigned“ einzustellen, wenn man Skripts verwenden möchte. In Bezug auf die Benutzbarkeit (in Verbindung mit einer noch akzeptablen Sicherheit) empfehle ich „RemoteSigned“.

Funktionen und Skripts

Eine Skriptdatei lässt sich mit jedem beliebigen Editor erstellen und hat die Dateiendung „.ps1“. In Skripts können aufeinanderfolgende Befehle folgen, aber auch Funktionen enthalten sein. Skripts werden von oben nach unten ausgeführt. Funktionen werden nur durch Aufruf dieser ausgeführt. Ein Skript das eigentlich nur eigene definierte Funktionen enthält die beim Start der PowerShell geladen werden und später aufgerufen werden können, nennt sich „profile.ps1“ und muss im Verzeichnis „WindowsPowerShell“ stehen. Der Pfad an dem dieses Verzeichnis liegen hängt vom Betriebssystem ab. In Windows XP muss es im Verzeichnis „Eigene Dateien“ liegen und in Vista im Verzeichnis „Dokumente“ des Userverzeichnisses.

Wenn wir beispielsweise folgendes in unsere „profile.ps1“ schreiben

```
function Get-Cksum ($file)
{
    $sum=0
    get-content -encoding byte -read 10kb $file | %{
        foreach ($byte in $_) { $sum += $byte }
    }
    $sum
}
```

dann wird beim Starten der PowerShell die Funktion Get-Cksum (Funktion zum Erstellen einer Prüfsumme) geladen. Diese Funktion erwartet einen Parameter „$file“ (, der in der PowerShell einfach vom Befehl durch ein Leerzeichen getrennt übergeben werden kann:

```
Get-Cksum meineDatei.txt
```

Man könnte diese Funktion auch in eine andere Skriptdatei packen und darin aufrufen. In der PowerShell müsste man dann nur noch den Namen des Skripts angeben das diese Funktion aufruft und eventuell weiterbearbeitet:

```
function Get-Cksum ($file)
{
    $sum=0
    get-content -encoding byte -read 10kb $file | %{
        foreach ($byte in $_) { $sum += $byte }
    }
    return $sum
}
```

```
$sum = Get-Cksum("meineDatei.txt")

echo "Prüfsumme von meineDatei.txt: $sum"
```

Wenn wir diesen Code in eine Datei „MeineDateiCksum.ps1“ schreiben, können wir das Skript wie folgt aufrufen

```
.\ MeineDateiCksum.ps1
```

und erhalten folgende Ausgabe (die im Beispiel ausgegebene Zahl hängt natürlich von der jeweiligen Datei ab)

```
Prüfsumme von meineDatei.txt: 53310
```

Natürlich können auch Skriptdateien mit einem Parameter aufgerufen werden. Die Variable $args enthält alle von der PowerShell an das Skript übergebenen Parameter. Es ist also ein Array, dass wie in dem Abschnitt über Arrays beschrieben in der Skriptdatei verwendet werden kann. So lässt sich die Anzahl übergebener Parameter beispielsweise mit $args.Length bestimmen. Auf einzelne Parameter kann mit dann per Indexoperator (z.B. $args[0]) zugegriffen werden oder man benutzt das $args-Array beispielswiese in einer Schleife um die übergebenen Parameter zu prüfen und/oder zu bearbeiten.