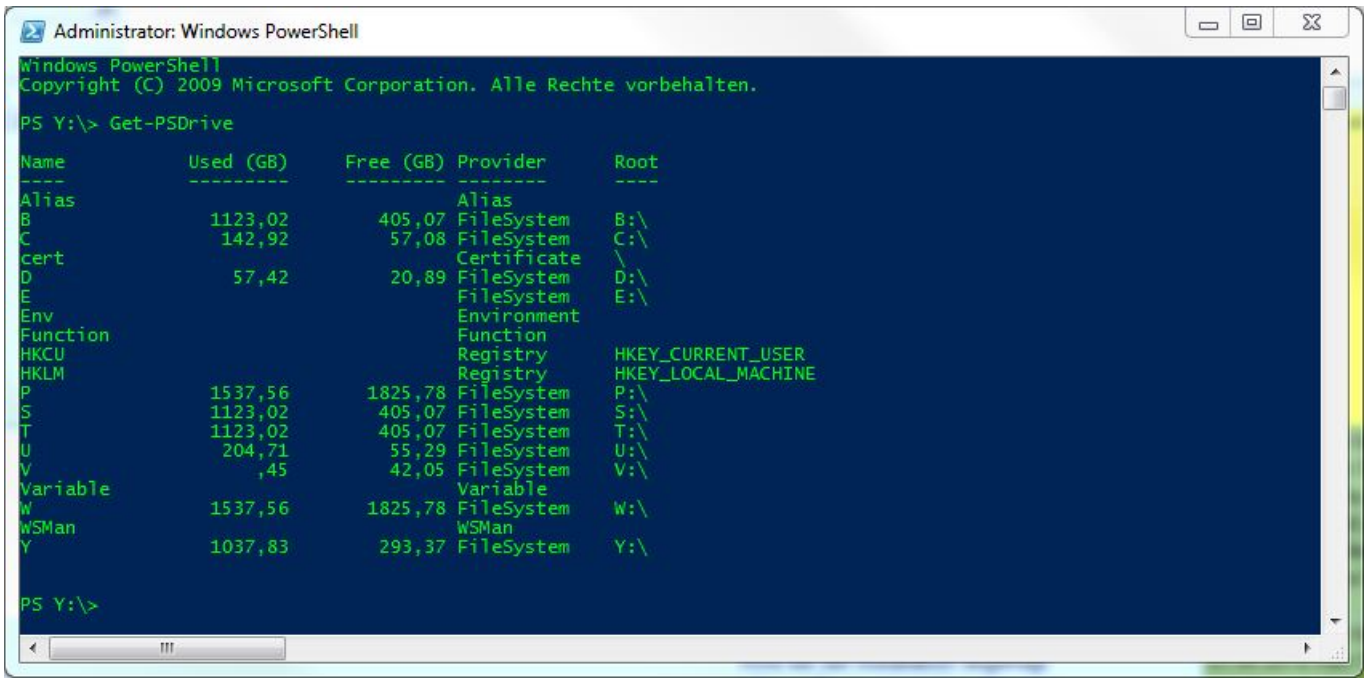
PowerShell

The Cook Book

Uwe Schimanski

Seab@er Software

Goch, den 14. Juli 2019



Published by Publisher

seabaer-ag.de

First printing, May 2018

Inhaltsverzeichnis

Vorwort

Im Jahr 1995 habe ich mir meinen ersten PC gekauft und dieser war mit dem Betriebssystem Windows 3.11 ausgestatet. Dort habe ich dann angefangen mit der Batch Programmierung. In der folgezeit kamen immer weitere Windows Versionen auf dem Markt und als Nachfolger der Dos-Shell kam dann die PowerShell.

Die Aufgaben für die Programmierung wurden immer anspruchsvoller und man kam dann an die Grenzen der Batch Programmierung. Nun habe ich mich in die PowerShell Programmierung gestürzt.

In diesem Handbook möchte ich die Grundlagen der PowerShell vorstellen und auch konkrete Lösungsbeispiele zeigen. Viel Spaß bei dem Lesen dieser Lektüre.

Als OS Systeme wurden von mir Windows 7 Enterprice und Windows 10 Pro verwendet.

Bei Fragen und Anregungen bin ich unter der folgenden Mail Adresse zu erreichen:
uws@seabaer-ag.de

Kapitel 1

Basis

1.1 Profile

Es gibt insgesamt 4 Profil Dateien, die für unterschiedliche User geladen werden. In der nachfolgenden Tabelle werden sie aufgelistet.

User	Pfad
Aktuellen User	`$Env:`Userprofile\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
	`$Env:`Userprofile\Documents\WindowsPowerShell\profile.ps1
Alle User	`$Env:`Systemroot\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
	`$Env:`Systemroot\System32\WindowsPowerShell\v1.0\profile.ps1

Tabelle 1.1: Liste Profil Datei

In der PowerShell ISE werden nur die `profile.ps1` Dateien geladen. In einer normalen PowerShell werden alle Profil Dateien geladen und zwar in der Reihenfolge:

- $Env:Systemroot\System32\WindowsPowerShell\v1.0\profile.ps1
- $Env:Systemroot\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
- $Env:Userprofile\Documents\WindowsPowerShell\profile.ps1
- $Env:Userprofile\Documents\WindowsPowerShell\Microsoft.PowerSell_profile.ps1

Eine leere Profil Datei kann man sich mit dem nachfolgenden Befehl erstellen. Das Verzeichnis WindowsPowerShell im Documents Ordner wird mit dem Befehl auch angelegt, wenn es nicht schon vorhanden ist.

```
PS D:\>New-Item -path $profile -itemtype file -force

PS D:\>New-Item -path $env:windir\System32\WindowsPowerShell\v1.0\profile.
   ps1 -itemtype file -force
```

Listing 1.1: Leere Profil Datei

1.2 Execution Policy

Ein PowerShell Script kann nur ausgeführt werden, wenn die Execution Policy auf `AllSigned, RemoteSigned` oder `Unrestricted` steht.

```
PS D:\>Get-ExecutionPolicy
Restricted

PS D:\>Get-ExecutionPolicy -List
Scope                          ExecutionPolicy
-----                          ---------------
MachinePolicy                  Undefined
UserPolicy                     Undefined
Process                        Undefined
CurrentUser                    Undefined
LocalMachine                   Restricted

PS D:\>Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine
Ausfuehrungsrichtlinie aendern
Die Ausfuehrungsrichtlinie traegt zum Schutz vor nicht vertrauenswuerdigen
   Skripts bei. Wenn Sie die Ausfuehrungsrichtlinie aendern, sind Sie
   moeglicherweise den im Hilfethema "about_Execution_Policies"
   beschriebenen Sicherheitsrisiken ausgesetzt. Moechten Sie die
   Ausfuehrungsrichtlinie aendern?
[J] Ja  [N] Nein  [H] Anhalten  [?] Hilfe (Standard ist "J"):
```

Listing 1.2: Execution Policy

1.3 Kommentare

Einzeilige Kommentare werden in der PowerShell mit einem # gemacht. Mehrzeilige Kommentare werden durch einen <# eingeleitet und mit einem #> abgeschlossen.

```
# Das ist ein einzeiliger Kommentar

<# Hier
koennen mehrere
Zeilen
stehen #>
```

Listing 1.3: Kommentare

1.4 Hilfe

1.4.1 Anzeigen

Eine Hilfe zu den Befehlen kann man sich mit `Get-Help <Cmdlet>` sich anzeigen lassen. Gibt man noch die Option `-full` mit an, so bekommt man eine Ausführlichere Hilfe angezeigt. Mit der Option `-example` werden Beispiele angezeigt. Hilfethemen werden in der PowerShell mit `Get-Help about_<Thema>` ausgegeben.

```
PS D:\>Get-Help Get-Date [-full] [-example]

PS D:\>Get-Help about_* # Anzeige aller Themen

PS D:\>Get-Help about_arrays
```

Listing 1.4: Hilfe anzeigen

1.4.2 Get-Help in Skripten

In einem Skript kann eine Kommentarbasierende Hilfe eingebaut werden. Diese Hilfe steht in einem Kommentarblock und besteht aus Schlüsselwörtern. Alle Zeilen dieser Hilfe müssen zusammenhängend geschrieben werden. Allen Schlüsselwörtern wird ein Punkt (.) vorangestellt. Die Hilfe kann für das Skript und auch für die Funktionen genutzt werden. Der Kommentarblock muss mindestens ein Schlüsselwort enthalten. Es gibt aber auch Schlüsselwörter, wie z.B. Example, die können in einem Kommentarblock mehrmals vorkommen. Die Hilfe wird dann folgendermaßen aufgerufen.

```
PS D:\>Get-Help C:\Work\MountShare.ps1 [-full] [-detailed] [-example]
```

Listing 1.5: Beispiel Get-Help

Ein Kommentarblock für Funktionen kann an drei Stellen stehen.

- Am Anfang der Funktion
- Am Ende der Funktion
- Vor der Funktion, aber es dard höchstens eine Leerzeile vor der Funktion sein

Der Kommentarblock für Skripte kann an drei Stellen stehen.

- Am Anfang des Skripts
- Am Ende Ses Skripts
- Irgendwo im Skript. Hier müssen zwischen dem Kommentarblock und der Anweisung mindestens zwei Leerzeilen eingefügt sein, sonst wird der Kommenatrblock als Hilfe für die Funktion interpretiert.

In der nachfolgenden Tabelle sind einige der Schlüsselwörter aufgelistet. Eine vollständige Liste erhält man mit dem Aufruf in einer PowerShell mit: `Get-Help about comment_based_help`.

Schlüsselwort	Beschreibung
.SYNOPSIS	Eine kurze Beschreigung des Skripts
.DESCRIPTION	Hier steht eine ausführliche Beschreibung des Skripts
.PARAMETER <Name>	Beschreibung eines Parameters. Kann mehrmals gesetzt werden
.EXAMPLE	Beispiele für den Befehl. Kann mehrmals gesetzt werden.
.INPUTS	Beschreibung der Eingabeobjekte
.OUTPUTS	Beschreibung der Ausgabeobjekte
.NOTES	Informationen über das Skript, z.B. Author, Mail, Version, ...
.LINK	Linksangaben. Kann mehrmals gesetzt werden

Tabelle 1.2: Schlüsselwörter

Die Kommentarbasierende Hilfe erstellt auch automatisch Ausgaben, wie der Name, die Syntax, die Parameterliste, die Parameter Attributtabelle, allgemeine Parameter und Hinweise. Hier ein Beispiel eines Hilfe Blocks.

```
<#
.SYNOPSIS
    Beschreibung des Skripts
.DESCRIPTION
    Hier kann eine ausfuehrliche Beschreibung stehen
.NOTES
    Author:  Paulchen Panther
    Mail:    p.panther@web.de
    Version: 19.02.25
.EXAMPLES
    Skript <Para1> <Para2>
```

```
#>
```

Listing 1.6: Beispiel Hilfe Block

1.5 Ausgabe Shell

Eine Ausgabe in der PowerShell wird mit dem Befehl `Write-Host` gemacht. Man kann auch die Text Farbe und Hintergrund Farbe definieren. Mit ForegroundColor (fore) wird die Text Farbe definiert und mit BackgroundColor die Hintergrund Farbe.

```
Write-Host -fore yellow -back blue "Text in der Farbe Gelb und der
   Hintergrund ist Blau."
```

Listing 1.7: Ausgabe Shell

Folgende Farben gibt es für ForegroundColor und BackgroundColor.
Black, DarkBlue, DarkGreen, DarkCyan, DarkRed, DarkMagenta, DarkYellow, Gray, DarkGray, Blue, Green, Cyan, Red, Magenta, Yellow, White
Möchte man Variablen formatiert ausgegeben, so geschieht das in dem Text mit Platzhaltern. Diese Platzhalter werden von 0 an aufwärtz gezählt. Ein Platzhalter steht immer in geschweifte Klammern und die Variable wird mit dem Parameter `-f` angegeben.

```
PS D:\>$FreeC = (Get-PSDrive).Free
PS D:\>$T1="Freier Platz auf dem Laufwerk C: {0:n2}" -f $FreeC
PS D:\Write-Host "$T1"
Freier Platz auf dem Laufwerk C: 906.027.008

PS D:\>$UsedC = (Get-PSDrive).Used
PS D:\>$SumC = $FreeC + $UsedC
PS D:\>$T2 = "Auf dem Laufwerk C: sind {0:n2} Bytes von {1:n0} Bytes belegt
   ({2:p})" -f $UsedC, $SumC, ($UsedC/SumC)
PS D:\Write-Host "$T2"
Auf dem Laufwerk C: sind 80.265 MB von 96.005 MB Belegt (83,61%)
```

Listing 1.8: Formatierte Ausgabe

Folgende Formatierungszeichen gibt es.

Formatierungszeichen	Beschreibung
c (oder C)	Währungsformat
d	Kurzes Datumsformat (Tag.Monat.Jahr vierstellig)
D	Langes Datumsformat (z.B. Montag, 21. Dezember 2009)
ddd	Wochentag kurz (z.B. Di)
dddd	Wochentag lang
e (oder E)	Wissenschaftliche Darstelllung einer Zahl
HH	Stunden im 24-Stunden-Format
hh	Stunden im 12-Stunden-Format
mm	Minuten
MM	Moant als Zahl
MMM	Monat kurz (z.B. Jan)
MMMM	Monat ausgeschrieben
n	Numerischer Wert mit Nachkommastellen, die Anzahl der Nachkommastellen erfolgt hinter dem n, (z.B. n2 für 2 Nachkommastellen)
p (oder P)	Prozentangabe
s	Datum und Zeit, getrennt durch dem Buchstaben T
ss	Sekunden
t	Kurzes Zeitformat (Stunde:Minute)
T	Langes Zeitformat (Stunde:Minute:Sekunde)

Continued on next page

continued from previous page.

Formatierungszeichen	Beschreibung
x (oder X)	Als Hexadezimal
Y	Monat voll ausgeschrieben und das Jahr vierstellig
yy	Jahreszahl zweistellig
yyyy	Jahreszahl vierstellig

Tabelle 1.3: Formatierungszeichen für den f-Operator

1.6 Shell Info

Informationen über die aktuelle PowerShell Session kann man sich mit `Get-Host` ausgeben lassen. Die aktuellen Einstellungen können auch verändert werden, wie zum Beispiel der WindowsTitle. Dieser Wert wird mit (Get-Host].UI.RAWUI.WindowsTitle="env:username ; PowerShell"neu gesetzt.

```
PS D:\>(Get-Host).UI.RAWUI
ForegroundColor         : DarkYellow
BackgroundColor         : DarkMagenta
CursorPosition          : 1,103
WindowPosition          : 0,54
CursorSize              : 25
BufferSize              : 120,3000
WindowSize              : 120,50
MaxWindowSize           : 120,81
MaxPhysicalWindowsSize  : 160,81
KeyAvailable            : False
WindowTitle             : Administrator: Windows PowerShell
```

Listing 1.9: Aktuelle Session

1.7 List Commands

In den nachfolgenden Beispielen wird gezeigt, wie man sich die Verfügbaren Kommandos in der PowerShell sich ausgeben kann.

```
PS D:\>Get-Command

CommandType     Name
-----------     ----
Alias           %
Alias           ?
Function        A:
Alias           ac
Cmdlet          Add-Computer
Cmdlet          Add-Content
Cmdlet          Add-History
Cmdlet          Add-Member
Cmdlet          Add-PSSnapin
Cmdlet          Add-Type

PS D:\>Get-Command -ListImported

CommandType     Name
-----------     ----
Cmdlet          Add-Content

PS D:\>Get-Command -Type Cmdlet
```

```
CommandType     Name
-----------     ----
Cmdlet          Add-Computer
Cmdlet          Add-Content
Cmdlet          Add-History
Cmdlet          Add-Member
Cmdlet          Add-PSSnapin
Cmdlet          Add-Type
```

Listing 1.10: Liste Kommandos

Welche Methoden ein Cmdlet enthält, kann man sich mit `Get-Member` sich anzeigen lassen.

```
PS D:>Get-Date | Get-Member

   TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         System.DateTime Add(System.TimeSpan
   value)
AddDays              Method         System.DateTime AddDays(double value)
AddHours             Method         System.DateTime AddHours(double value)
AddMilliseconds      Method         System.DateTime AddMilliseconds(double
   value)
AddMinutes           Method         System.DateTime AddMinutes(double value)
AddMonths            Method         System.DateTime AddMonths(int months)
AddSeconds           Method         System.DateTime AddSeconds(double value)
.
.
```

Listing 1.11: Anzeige Methoden

1.8 Datum und Zeit

Ein Datum und Uhrzeit kann man mit `Get-Date` sich holen. Mit der Option `-Format` kann man die Ausgabe formatieren. In der nächsten Tabelle sind die Formatierungsmöglichkeiten aufgelistet.

Format	Beschreibung
d	Datum kurz
D	Datum lang
f	Datum mit Uhrzeit kurz
F	Datum mit Uhrzeit lan
g	General Date mit Uhrzeit kurz
G	General Date mit Uhrzeit lang
m,M	Monat/Tag
o,O	Datum/Uhrzeit
r,R	RFC1123
s	Sortable Datum/Uhrzeit
t	Uhrzeit kurz
T	Uhrzeit lang
u	Universal sortable Datum/Uhrzeit
U	Universal Datum/Uhrzeit
y,Y	Jahr Monat

Tabelle 1.4: Get-Date Formatierung

Hier sind einige Beispiele für die Datum / Uhrzeit Formatierung.

```
$MyColor='Gray'
$MyDate=Get-Date
Write-Host -ForegroundColor $MyColor "              Datum kurz: $(Get-Date -
    Format d)"
Write-Host -ForegroundColor $MyColor "              Datum kurz: $(Get-Date -
    Format dddd)"
Write-Host -ForegroundColor $MyColor "              Datum lang: $(Get-Date -
    Format D)"
Write-Host -ForegroundColor $MyColor "        Datum formatiert: $(Get-Date -
    Format yyyy_MM.dd)"
Write-Host -ForegroundColor $MyColor "  Datum mit Uhrzeit kurz: $(Get-Date -
    Format f)"
Write-Host -ForegroundColor $MyColor "  Datum mit Uhrzeit lang: $(Get-Date -
    Format F)"
Write-Host -ForegroundColor $MyColor "General mit Uhrzeit kurz: $(Get-Date -
    Format g)"
Write-Host -ForegroundColor $MyColor "General mit Uhrzeit lang: $(Get-Date -
    Format G)"
Write-Host -ForegroundColor $MyColor "      Datum nach RFC1123: $(Get-Date -
    Format R)"
Write-Host -ForegroundColor $MyColor "               Monat/Tag: $(Get-Date -
    Format m)"
Write-Host -ForegroundColor $MyColor "               Monat/Tag: $(Get-Date -
    Format M`n)"
Write-Host -ForegroundColor $MyColor "  Tag aus einer Variable: $($MyDate.
    Day)"
Write-Host -ForegroundColor $MyColor "Monat aus einer Variable: $($MyDate.
    Month)"
Write-Host -ForegroundColor $MyColor " Jahr aus einer Variable: $($MyDate.
    Year)`n"
Write-Host -ForegroundColor Green "Aktueller Tag +20: $((Get-Date).AddDays
    (20).DayOfWeek)"
Write-Host -ForegroundColor Green "(Get-Date).ToShortTimeString(): $((Get-
    Date).ToShortTimeString()) "
Write-Host -ForegroundColor Green "(Get-Date).ToShortDateString(): $((Get-
    Date).ToShortDateString()) "
Write-Host -ForegroundColor Green "Auflistung aller Methoden"
Write-Host -ForegroundColor Green "-------------------------"
Get-Date|Get-Member

PS D:\>.\ExampleDatumFormat.ps1
              Datum kurz: 03.05.2019
              Datum kurz: Freitag
              Datum lang: Freitag, 3. Mai 2019
        Datum formatiert: 2019_05.03
  Datum mit Uhrzeit kurz: Freitag, 3. Mai 2019 06:57
  Datum mit Uhrzeit lang: Freitag, 3. Mai 2019 06:57:52
General mit Uhrzeit kurz: 03.05.2019 06:57
General mit Uhrzeit lang: 03.05.2019 06:57:52
      Datum nach RFC1123: Fri, 03 May 2019 06:57:52 GMT
               Monat/Tag: 03 Mai
               Monat/Tag: 5

  Tag aus einer Variable: 3
Monat aus einer Variable: 5
 Jahr aus einer Variable: 2019

 Aktueller Tag +20: Thursday
(Get-Date).ToShortTimeString(): 06:57
(Get-Date).ToShortDateString(): 03.05.2019
Auflistung aller Methoden
```

```
-------------------------


   TypeName: System.DateTime

Name                  MemberType      Definition
----                  ----------      ----------
Add                   Method          System.DateTime Add(System.TimeSpan
    value)
AddDays               Method          System.DateTime AddDays(double value)
AddHours              Method          System.DateTime AddHours(double value)
AddMilliseconds       Method          System.DateTime AddMilliseconds(double
    value)
AddMinutes            Method          System.DateTime AddMinutes(double value)
AddMonths             Method          System.DateTime AddMonths(int months)
AddSeconds            Method          System.DateTime AddSeconds(double value)
.
.
```

Listing 1.12: Beispiele Datum / Uhrzeit

1.9 Variablen

1.9.1 Definition

Variablen werden in der PowerShell mit einem `$-Zeichen` vor dem Namen definiert. Eine Deklaration eines Datentyps muss man nicht machen, das erfolgt automatisch. Man kann aber trotzdem ein Datentyp angeben. Hierzu wird vor der Variable der Datentyp mit angegeben.

```
PS D:\>$MyVar = "String Variable"

PS D:\>$MyInt = 2

PS D:\>$a, $b, $c = 5, 6, 7

PS D:\>$a = 3 * 4

PS D:\>$d = $e = $f = 5

PS D:\>[Int]$MyNumber = Read-Host "Bitte eine Zahl eingeben: "

PS D:\>$MYDrive = "L","P","M" # array
PS D:\>$MyShare = @("\\server1\daten","\\server1\public","\\server1\
   multimedia")
```

Listing 1.13: Variablen Beispiele

In der nächsten Tabelle werden die gängigsten Variablen aufgelistet.

Datentyp	Beschreibung
[Array]	Array definieren
[Bool]	True oder False
[byte]	vorzeichenloses 8 Bit Zeichen
[char]	Ein Zeichen
[DateTime]	Datum und Uhrzeit
[Decimal]	128-bit dezimal Wert
[Double]	Doppelte Genauigkeit, Gleitkommazahl
[Guid]	Global eindeutige 32 Byte ID
[HashTable]	Hash-Tabelle
[Int32], [Int]	32-bit Integer
[long]	64-bit Integer
[PsObject]	PowerShell Object
[Regex]	Regular expression
[ScriptBlock]	PowerShell script Block
[Single], [Float]	Einfache Genauigkeit, Gleitkommazahl
[String]	Zeichenkette
[Switch]	PowerShell Switch-Parameter
[TimeSpan]	Zeitintervall
[XmlDocument]	XML-Dokument

Tabelle 1.5: Datentypen

Werden die Variablen in einem eigenen `Param-Block` definiert, so werden sie bei der `Get-Help` Funktion mit angezeigt un sie können als Parameter bei dem Aufruf des Programms mit übergeben werden.

```
PARAM
(
 $StrAuthor = "Harry Hirsch"
 [string]$StrCompany = "Seab@er Software"
 [init]$Wait = 5
```

```
)

PS D:\>ParamBlock.ps1 -StrAuthor "Anton Meise" -Wait 10
```

Listing 1.14: Param-Block

1.9.2 Vordefinierte Variablen

In der PowerShell gibt es eine Menge von vordefinierten Variablen. Diese sind in der nächsten Tabelle aufgelistet.

Variable	Beschreibung
`$_`	Das aktuelle Objekt einer Pipeline, Filter oder Schleifen. In einer Try/Catch Schleife enthält die Variable den Fehler.
`$$`	Den letzten Token der Shell-Eingabe
`$?`	Exit Code des Befehls
`$Args`	Übergebene Parameter
`$Error`	Die letzten 256 Fehlermeldungen
`$Home`	Das Home-Vereichnis des Benutzers
`$Input`	Input pipe einer Funktion oder Codeblocks
`$Match`	Hashtabelle, die alle Elemente enthält, welche durch dem `-match` Operator gefunden wurde
`$MyInvocation`	Informationen über das aktuelle Skript oder der Kommandozeile
`$Host`	Informationen über den Computer
`$LastExitCode`	Exit Code eines nicht PowerSehll Befehls
`$True`	Wahr
`$False`	Falsch
`$Null`	Null-Objekt
`$OFS`	Output Fild Operator, also das Trennzeichen
`$ShellID`	Ausgabe des Shell Namen
`$StackTrace`	Stacktrace der letzten Aktion
`$env:variablename`	Umgebungsvariablen
`$global:variablename`	Variable `global` setzen und lesen

Tabelle 1.6: Vordefinierte Variablen

1.10 Mehrzeiliger Befehl

Einen mehrzeiligen Befehl in einem Skript wird mit einem Rückwärts geneigtes Hochkomma, `Backquote (‘)`, gemacht.

```
Write-Host -Foreground Gray ‘
"Hier koente ein Text stehen"
```

Listing 1.15: Mehrzeilge Kommandos

1.11 Sleep

Möchte man ein Script eine bestimmte Zeit pausieren lassen, so kann das mit `Start-Sleep` gemacht werden. Als Parameter wird die Zeiteinheit angegeben. Es gibt die Parameter `-Milliseconds` und `Seconds`.

```
PS D:\># Wartet eine 1/2 Sekunde
PS D:\>Start-Sleep -Milliseconds 500

PS D:\># Wartet 3 Sekunden
PS D:\>Start-Sleep -Seconds 3

PS D:\># Fortschrittanzeige
PS D:\>for ($i=1; $i -lt 10; $i++) {Write-Host -Fore Gray "." -NoNewline
>> Start-Sleep -s 1}
>> Write-Host "‘n"
>>
.........
PS D:\>
```

Listing 1.16: Beispiele

1.12 Externe Programme

Muss in einem Skript ein externes Programm ausgeführt werden, so wird dem Programmaufruf ein `&` vorangestellt.

```
& $Taskrunner /i $ScriptPath$E /of $NwdPath$A /log $LogFilePath$B
```

Listing 1.17: Beispiel

1.13 Module laden

Hat man eigene Functions in einer `psm1` Datei ausgegliedert und möchte sie nun in einem anderen Skript laden, so geschieht das mit `Import-Module`. Die Angabe des Pfades zu der Datei ist nötig, ausser man legt die Datei in den Modul Suchpaden ab. Diese können mit `$env:psmodulepath` ausgegeben werden.

```
PS D:\>Import-Module "D:\Testing\WriteLog.psm1" -force

PS D:\>$env:psmodulepath
C:\Users\schimanu\Documents\WindowsPowerShell\Modules;
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\;
C:\Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\;
C:\Program Files\WindowsPowerShell\Modules\
```

Listing 1.18: Load Module

Eine Auflistung, welche Functions / Module zu Verfügung stehen, kann man mit `Get-Module` und `Get-Command` sich anzeigen lassen.

```
PS D:\>Get-Module -ListAvailable

ModuleType Name                         ExportedCommands
---------- ----                         ----------------
Manifest   AppLocker                    {}
Manifest   BitsTransfer                 {}
Manifest   PSDiagnostics                {}
Manifest   TroubleshootingPack          {}
Manifest   WebAdministration            {}
Manifest   AutoItX                      {}
Manifest   SQLASCMDLETS                 {}
Manifest   SQLPS                        {}
Manifest   SqlServer                    {}

PS D:\>Get-Command -CommandType "Function"

CommandType     Name
-----------     ----
Function        A:
Function        B:
Function        C:
Function        cd..
Function        cd\
Function        Clear-Host
Function        CreateCredentialObject
Function        CreatePasswordFile
.
.
```

Listing 1.19: Show Module

Eine Function kann man mit `Remove-Item` wieder aus der Shell entfernen.

```
PS D:\>Remove-Item -Path Function:CreatePasswordFile
```

Listing 1.20: Unload Function

Kapitel 2

Advanced

2.1 Vergleichs Operatoren

Folgende Vergleichs Operatoren gibt es fÃ¼r die PowerShell. Soll die Groß- und Kleinschreibung bei dem vergleichen auch berücksichtigt werden, so wird noch ein c vor dem Operator gesetzt. Wie zum Beispiel ein -ceq oder -cne.

Operator	Beschreibung
-eq	gleich
-ne	nicht gleich
-lt	kleiner
-le	kleiner oder gleich
-gt	größer
-ge	größer oder gleich
-like	Suchmuster mit Wildcards
-contains	Array prüfen
-notcontaints	Array prüefen
-match	Suchmuster ohne Wildcards
-nomatch	Suchmuster ohne Wildcards

Tabelle 2.1: Vergleichs Operatoren

```
PS D:\>"Schwarzwald" -like "Schw*" # gibt true zurueck

PS D:\>$c = "Green","Yellow","Red"
PS D:\>$c -containts "Blue" # Gibt false zurueck, da nicht im array
    vorhanden

PS D:\>"Schwarzwald" -match "rzw" # gibt true zurueck
```

Listing 2.1: Vergleich Beispiele

2.2 If Abfrage

```
$MYDrive = "L","P","M"
$MyShare = @("\\server1\daten","\\server1\public","\\server1\multimedia")

for ($i=0; $i -lt $MyShares.Length; $i++)
{
  Log -text ("Mount Share " + $MyShares[$i] + " with Drive " + $MyDrive[$i])
      -severity 1
  New-PSDrive -Name $MyDrive[$i] -PSProvider FileSystem -Root $MyShares[$i]
     -Persist
}

if ($FileExist -eq "true")
{
  Write-Host -fore green "‘t File copy to destination."
  Copy-Item -Path D:\tmp\uws.txt -Destination d:\work\uws\uws.txt -force
}
else
{
 Write-Host -fore red "‘t Can’t copy the file.‘n"
}
```

Listing 2.2: Beispiel

Möchte man ermitteln, ob es ein Verzeichnis, Datei oder eine Variable gibt, so kann das mit `Test-Path` gemacht werden.

```
PS D:\>Test-Path env:tmp
True

PS D:\>Test-Path "HKLM:\SOFTWARE\Microsoft\Windows\System64"
False

PS d:\>cat DemoTestPath.ps1
if (Test-Path -eq "readme.txt")
{
  Write-Host "Datei vorhanden"
}

$MyFile = Test-Path Docs\readme.txt

if (Test-Path .\Dokumente\Version) {Get-Content .\Dokumente\Version}

# Pruefen, ob verschiedene Dateien vorhanden sind. Der Stern nach der Angabe
    des
# Verzeichnis ist wichtig und muss gesetzt werden. Ausser include gibt es
   auch
# exclude.
Test-Path .\Dokumente\* -include *.txt,*.pdf

# Nach Verzeichnis oder Datei pruefen, das geschieht mit -pathType.
# container => Verzeichnis und leaf => Datei
Test-Path .\Dokumente -pathType container
```

Listing 2.3: Test-Path

2.3 Function

Eine Function wird mit dem Schlüsselwort function <NameFunction> definiert. Es können Übergabe Variablen definiert werden. Die Angabe der Namen der Variablen für die Function kann auch weggelassen werden, dann müssen sie aber in der richtigen Reihenfolge stehen.

```
function CreateCredentialObject ($FilePath, $Username)
{
  if (Test-Path $FilePath)
  {
    # Load password file
    $Password = cat $FilePath | ConvertTo-SecureString

    # Create credential object
    Write-Host "Creating credential object" -ForegroundColor Green
    $Cred = New-Object -typename System.Management.Automation.PSCredential -
       ArgumentList $Username, $Password
    Return $Cred
  }
  else
  {
    Write-Host "Password file missing" -ForegroundColor Red
  }
}

# Angabe -FilePath und -Username optional
$MyPasswd = CreateCredentialObject -FilePath "Y:\Daten\Scripts\PowerShell\
   passwd.dat" -Username "schimanu"
```

Listing 2.4: Function Beispiel

Möchte man eine Funktion erstellen, die Pipeline-Inhalte verarbeiten kann, so muss der Aufbau einen speziellen Charakter haben. Die Funktion wird in drei (optinonale) Bereiche unterteilt. Die Namen der Bereiche sind begin, process und end. Nachfolgend ein Beispiel, wie eine Zeichenkette in einer Binärzahl umgewandelt wird.

```
function Convert-String2Bin
{
 param([Parameter(ValueFromPipeline=$true)][string]$Zahl)
 # Im begin Teil steht die Pipeline nicht zu Verfuegung
 begin
 {
  $summe=0
 }
 process
 {
  $Laenge = $Zahl.Length-1
  $Zahl.ToCharArray() | ForEach-Object {
   $summe += [byte]$_.ToSttring()*[System.Math]::pow(2,$Laenge)
   $Laenge--
  }
 }
 end { return $summe }
}

PS D:\>Convert-String2Bin -Zahl 10101010

PS D:\>10101010 | Convert-String2Bin
```

Listing 2.5: Beispiel Pipeline Funktion

2.4 String Manipulation

In diesem Abschnitt wird beschrieben, wie man Strings manipulieren kann. Als erstes wandeln wir den Text auf Großbuchstaben um und dann auf Kleinbuchstaben.

```
PS D:\>$("powersehll").ToUpper()
POWERSHELL

PS D:\>$("POWERSHELL").ToLower()
powershell
```

Listing 2.6: Groß / Klein

Nun prüfen wir, ob eine Zeichenfolge in dem Text vorhanden ist.

```
PS D:\>$Var1="PowerShell"
PS D:\>$("$Var1").Contains("wer")
True
```

Listing 2.7: String prüfen

Als nächstes prüfen wir, ob der String mit einer Zeichenfolge beginnt oder endet.

```
PS D:\>$("$Var1").StartsWith("Pow")
True

PS D:\>$("$Var1").StartsWith("pow")
False

PS D:\>$("$Var1").EndsWith("ell")
True
```

Listing 2.8: Start / End mit

Einen Text ersetzen in dem String wird folgendermaßen gemacht.

```
PS D:\>$("$Var1").Replace("Power","Bash")
BashShell
```

Listing 2.9: Ersetzen

Mittels des Befehls Substring kann der String zerlegt werden.

```
PS D:\># von Position
PS D:\>$("$Var1").Substring("2")
werShell

PS D:\># von Position und laenge
PS D:\>$("$Var1").Substring("2","5")
werSh

PS D:\>$("$Var1").Substring("","5")
Power
```

Listing 2.10: Substring

Nun kürzen wir den String ein.

```
PS D:\>$("Example.ps1").TrimStart("Exa")
mple.ps1

PS D:\>$("Example.ps1").TrimEnd(".ps1")
Example
```

Listing 2.11: String kürzen

2.5 Parameter Übergabe

Mit einem `param` Block kann man Parameter definieren, die übergeben werden können. Mit `CmdletBinding` kann in der Shell mit der Tab-Taste die definierten Parameter vervollständigt werden. Soll ein Parameter unbedingt angegeben werden, so setzt man das `Mandatory=$True`.

```
[CmdletBinding()] # In der Shell koennen die Parameter mit der Tab Taste
   vervollstaendigt werden.
Param (
  [Parameter(Mandatory=$True)] # Der naechste Parameter muss angegeben
     werden, sonst wird er abgefragt.
  [string]$Parameter1,
  [Parameter(Mandatory=$False)]
  [bool]$Parameter2 = $False,
  [int]$Parameter3 = 560,
  [switch]$Parameter4
)

# Ausgabe der Parameter
Write-Host " String Parameter: $Parameter1"
Write-Host "   Bool Parameter: $Parameter2"
Write-Host "Integer Parameter: $Parameter3"
Write-Host " Switch Parameter: $Parameter4"
```

Listing 2.12: Parameter übergabe

Dürfen nur bestimmte Werte der Variablen übergeben werden, so kann man das mit `ValidateSet` machen.

```
Function MyFunc
{
 Param (
  [ValidateSet(1,5,20)][int]$MyVal
 )
 Process
 {
  Write-Host "Value: $MyVal"
 }
}
```

Listing 2.13: ValidateSet

In der nächsten Tabellen werden die Parameterattribute und die Properties für die Parameter aufgelistet.

Property	Beschreibung
SupportsShouldProcess	$true, wenn die Parameter Confirm und WhatIf zur Verfügung stehen sollen und die Funktion eine Bestätigung anfordert.
ConfirmImpact	Legt den Bstätigungslevel fest
DefaultParameterSetName	Legt den Namen des Standardparametersets fest, wenn ein Parameter keinen Parameterset zugeordnet werden kann.

Tabelle 2.2: Prperties CmdletBinding

```
function MyFunc
{
  [CmdletBinding(SupportsShouldProcess=$true)]
  param ([string]$MyPath)
}
```

Listing 2.14: Beispiel CmdletBinding

Parameterattribut	Beschreibung
CmdletBinding	Die Bindung der einzelnen Parameter wird auf dieselbe Weise durchgeführt, wie bei einem Cmdlet.
Parameter	Legt erweiterte Eigenschaften fest
Alias	Gibt dem Parameter einen Kurznamen
AllowNull	Für einen Pflichtparameter kann ein $null-Wert übergeben werden, was sonst nicht möglich ist
AllowEmptyString	Wie AllowNull, aber als Zeichenfolge
AllowEmptyCollection	Wie AllowNull, hier alls Collection-Parameter
ValidateCount	Gibt die minimale und maximale Anzahl an Werten für diesen Parameter an
ValidateLength	Gibt die minimale und maximale Länge des Wertes für diesen Parameter an
ValidatePattern	Legt einen regulären Ausdruck an, der den Argumentenwert matchen muss. Ansonsten ist eine Fehlermeldung die Folge
ValidateRange	Gibt den minimalen und maximalen Wert für diesen Parameter an
ValidateScript	Legt über einen Skriptblock einen Ausdruck fest, der in Verbindung mit dem Parameterwert $true ergeben muss
ValidateNotNull	Für das Argument darf kein $null-Wert übergeben werden.
ValidateNotNullOrEmpty	Für das Argument darf kein $null-Wert oder eine leere Variable übergeben werden
ValidateSet	Legt eine Menge von Werten fest, von denen einer übereinstimmen muss

Tabelle 2.3: Parameterattribute

Property	Beschreibung
Mandatory	$true, wenn es ein Pflichtparameter ist
Position	Gibt die Position des Parameterwertes an, damit dieser dem Parameter zugeordnet werden kann, ohne das der Parametername benötigt wird
ParameterSetName	Gibt den Namen eines Parametersatzes an, zu dem der Parameter gehören soll
ValueFromPipeline	$true, wenn der Wert des Parameters auch aus der Pipeline geholt werden kann
ValueFromPipelineByPropertyName	$true, wenn der Wert des Parameters anhand des gleich lautenden Namens der Eigenschaft des Objekts in der Pipeline zugeordnet wird
ValueFromRemainingArgument	$true, wenn dem Parameter alle verbleibenden Werte der Befehlszeile zugeordnet werden, die nicht an andere Parameter gebunden sind
HelpMessage	Legt eine kurze Beschreibung des Parameters fest

Tabelle 2.4: Properties parameter-Attributes

Man kann auch die Parameter aus der Variable `$args` abfragen.

```
$i=1
ForEach ($arg in $args)
{
# Ausgabe der Parameter
  Write-Host (" Parameter $i" + ": $arg")
  $i++
}
```

Listing 2.15: Args Beispiel

2.6 Fehlerbehandlung

In der PowerShell gibt es zwei Typen von Fehlern. Die einen verursachen einen sofortigen Abbruch des Kommandos oder des Skripts (Terminating Errors) und die anderen erlauben eine Fortsetzung des Kommandos / Skripts (Non-Terminating Errors). Terminierte Fehler kann man in der PowerShell mit einem Try-Catch Block abfangen. Nicht terminierte Fehler kann man standard mäsig mit diesem Block nicht abfangen, es sei denn, Die Variable `$ErrorActionPreference` wird auf `Stop` gesetzt.

2.6.1 Terminierter Fehler

Wie oben schon erwähnt, kann man terminierte Fehler mit einem Try-Catch-Finally Block abfangen. Try-Catch muss definiert werden, wohin gehend der Finally Block optional ist. In dem Try-Block werden die Anweisungen geschrieben und in dem Catch-Block die Fehlerbehandlung. In dem Finally-Block werden Kommandos geschrieben, die in jedem Fall ausgeführt werden sollen. Tritt eine Exception auf, so wird der Fehlertext in der Variable `$_` abgespeichert.

```
Try
{
  Write-Host "Hier stehen Kommandos"
}
Catch
{
  Write-Host "Fehlerbehandlung"
  Write-Host " Exception: $_"
}
Finally
{
  Write-Host "Wird immer ausgefuehrt."
}
```

Listing 2.16: Try-Catch-Finally Block

Für einen Try-Block kann es auch mehrere Catch-Blöcke geben. Es muss nur zu dem Catch Befehl der Typ der Exception in echigen Klammern angegeben werden. Den Namen der Exception kann man aus der $Error-Variable auslesen: `$Error[0].Exception.GetType().FullName`.

```
Catch [System.Management.Automation.ItemNotFoundException]
```

Listing 2.17: Catch Block

Weitere Informationen über die erfolgte Exception kann man sich folgendermaßen ausgeben lassen.

```
PS D:\>$Error[0]
Ausnahme beim Aufrufen von "Execute" mit 3 Argument(en):  "Tabelle 'Users'
    ist bereits vorhanden."
Bei Y:\daten\Scripts\PowerShell\DemoAccessDB.ps1:32 Zeichen:17
+      $DB_Con.Execute <<<< ($Sql_Command)
    + CategoryInfo          : NotSpecified: (:) [],
        MethodInvocationException
    + FullyQualifiedErrorId : DotNetMethodException

PS D:\>$Error[0].Exception.GetType().Name
MethodInvocationException

PS D:\>$Error[0].Exception.GetType().FullName
System.Management.Automation.MethodInvocationException

PS D:\>$Error[0].Exception.Message
Ausnahme beim Aufrufen von "Execute" mit 3 Argument(en):  "Tabelle 'Users'
    ist bereits vorhanden."

PS D:\>$Error[0].InvocationInfo.ScriptName
```

```
Y:\daten\Scripts\PowerShell\DemoAccessDB.ps1

PS D:\>$Error.Count
29

PS D:\>$Error.Clear() # Inhalt loeschen
PS D:\>$Error.Remove($Error[$Error.count-1]) # letzten Eintrag loeschen
```

Listing 2.18: Exception Info

2.6.2 Nicht Terminierter Fehler

Nicht Terminierte Fehler kann man mit `-ErrorAction <Wert>` abfangen.

Wert	Abkürzung	Beschreibung
SilentlyContinue	-EA 0	Fehlermeldung wird unterdrückt und das Skript wird weiter ausgeführt
Stop	-EA 1	Fehlermeldung ausgeben und das Skript stoppen
Continue	-EA 2	Fehlermeldung ausgeben und das Skript wird weiter ausgeführt
Inquire	-EA 3	Fehlermeldung ausgeben und User fragen, ob das Skript weiter ausgeführt werden soll

Tabelle 2.5: Fehlercode

In dem nachfolgenden Beispiel wird die Ausgabe der Fehlermeldung nur für den einen Befehl unterdrückt.

```
PS D:\>Get-Process $Process -ErrorAction SilentlyContinue
```

Listing 2.19: Fehlermeldung unterdrücken

Möchte man die Fehlermeldung für das ganze Skript unterdrücken, so muss hierzu die Variable `$ErrorActionPreference` gesetzt werden.

```
PS D:\>$ErrorActionPreference="Continue"
```

Listing 2.20: Fehlermeldung abschalten

Kapitel 3

Lösungen

3.1 Ausgabe Shell und File

Einen Text kann man mit der folgenden Funktion in einer Datei schreiben und auch auf dem Screen ausgeben.

```
function Log($text,$errcode){
  $Time = Get-Date -f [dd.mm.yyyy_HHmmss]
  switch ($errcode)
  {
    0 {
       Write-Host -ForegroundColor green "$text"
       #add-content -path $Logfile -value "[INF] $Time $text"
            break
      }

   1 {
      $Global:BackupError = $true
      Write-Host -ForegroundColor Yellow "$text"
      #add-content -path $Logfile -value "[WAR] $Time $text"
      break
     }

   2 {
      $Global:BackupError = $true
      Write-Host -ForegroundColor Red "$text"
      #add-content -path $Logfile -value "[ERR] $Time $text"
      break
     }

   default {
     Write-Host "$text"
     #add-content -path $Logfile -value "[---] $Time $text"
     break
    }
  }
}

Log -text "Mounting Shares from File Server $Server" -errcode 0
```

Listing 3.1: Ausgabe Screen / Datei

3.2 Benötigte PowerShell Version

Wird eine bestimmte PowerShell Version für das Ausführen des Scriptes benötigt, so kann man das folgendermaßen machen.

```
$Global:MSG = "Zum Ausfuehren dieses Skripts ist mindestens PowerShell 3
    erforderlich!"

if ($PSVersionTable.PSVersion.Major -lt 3)
{
  Write-Warning -Message "$Global:MSG"
  #Write-Error -Message " ERROR-001: Other PowerShell Version found!" #
     Ueberschreibt die Error Meldung
  return
}
```

Listing 3.2: PowerShell Version

3.3 Ländereinstellung

Möchte man die aktuelle Ländereinstellung in Erfahrung bringen, so kann man das mit `get-UIculture` machen.

```
$MyCountry = get-UIculture

if ( $MyCountry = "de-DE")
{
  $Global:MSG = "Zum Ausfuehren dieses Skripts ist mindestens PowerShell 3
     erforderlich!"
}
elseif ( $MyCountry = "en-US")
{
  $Global:MSG = "At least PowerShell 3 is required to run this script!"
}
```

Listing 3.3: Ländereinstellung

3.4 Horizontale Linie

Möchte man eine horizontale Linie ausgeben und ein Zeichen dafür definieren, so kann man es folgendermaßen machen.

```
$MaxLineLength = 80
$HeadLine = New-Object -TypeName System.String "=", $MaxLineLength
Write-Host -ForegroundColor Red $HeadLine
```

Listing 3.4: Horizontale Linie

3.5 Passwort Eingabe mit Vergleich

Soll ein Passwort eingegeben und verglichen werden, so wird das in dem nächste Beispiel demonstriert. Nach erfolgreichem Vergleich wird das Passwort in einer Datei geschrieben.

```
function CreatePasswordFile ($FilePath)
{
  if (Test-Path $FilePath)
  {
    Write-Host "Password file allready exist" -ForegroundColor Green
  }
  else
  {
    Try
    {
      # Dialog to enter credentials
      Write-Host "Please type password."
      #Read-Host -Prompt "Enter Password" -AsSecureString | ConvertFrom-
         SecureString | Out-File $FilePath

      #New Version
      $passwd1=Read-Host -Prompt "   Enter Password" -AsSecureString
      $passwd2=Read-Host -Prompt "Re-Enter Password" -AsSecureString
      $pwd1 = [Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.
         InteropServices.Marshal]::SecureStringToBSTR($passwd1))
      $pwd2 = [Runtime.InteropServices.Marshal]::PtrToStringAuto([Runtime.
         InteropServices.Marshal]::SecureStringToBSTR($passwd2))
      if ("$pwd1" -cne "$pwd2")
      {
        Write-Host "Password not equal! Exit Program." -ForegroundColor Red
        exit 1
      }
      else
      {
        $passwd1 | ConvertFrom-SecureString | Out-File $FilePath
      }
    }
    Catch
    {
      Write-Host $_.Exception.Message -ForegroundColor Red
      break
    }
    Write-Host "Writing password file to" $FilePath -ForegroundColor Green
  }
}
```

Listing 3.5: Passwort Eingabe

3.6 Passwort Datei auslesen

Eine Passwort Datei kann man auslesen und dann ein Credential Object damit erstellen.

```
function CreateCredentialObject ($FilePath, $Username)
{
  if (Test-Path $FilePath)
  {
    # Load password file
    $Password = cat $FilePath | ConvertTo-SecureString

    # Create credential object
    Write-Host "Creating credential object" -ForegroundColor Green
    $Cred = New-Object -typename System.Management.Automation.PSCredential -
       ArgumentList $Username, $Password
    Return $Cred
  }
  else
  {
    Write-Host "Password file missing" -ForegroundColor Red
  }
}

CreatePasswordFile -FilePath ($env:AppData + "\" + $Server + ".dat")
$MyPasswd = CreateCredentialObject -FilePath ($env:AppData + "\" + $Server +
    ".dat") -Username "$env:Username"
```

Listing 3.6: Credential Object

3.7 Check Service

In dem Beispiel wird ein Dienst überwacht und wenn er nicht den Status Running hat, wird ein Dienst mit Restart gestartet und der Überwachte Service mit Start.

```
$LogFilePath = 'D:\Scripts\StartBentleyService\log\'
$LogDate = Get-Date -Format yyyy_MM_dd
$LogFileName = $($LogDate) + '_SelectServerGateway.log'
$LogFile = $LogFilePath + $LogFileName
$ServiceName = 'Bentley Select Server Gateway'
$SqlServiceName = 'SQL Server (SELECTSERVER)'
$text = "None"

# For debugging
Write-Host -ForegroundColor Gray "      LogfilePath: $LogFilePath"
Write-Host -ForegroundColor Gray "          LogDate: $LogDate"
Write-Host -ForegroundColor Gray "      LogFileName: $LogFileName"
Write-Host -ForegroundColor Gray "         Log File: $LogFile"
Write-Host -ForegroundColor Gray "     Service Name: $ServiceName"
Write-Host -ForegroundColor Gray " SQL Service Name: $SqlServiceName"

#----- Function ----------------------------------------------------------
#        Name: Log
# Description: Write Log File
#-------------------------------------------------------------------------
function Log($text,$severity){
  $MyDate = Get-Date -Format dd.MM.yyyy
  $MyTime = Get-Date -Format HH:mm:ss
  $Time = "[$MyDate $MyTime]"
  switch ($severity)
  {
    0 {
      Write-Host -ForegroundColor green "$text"
      add-content -path $Logfile -value "[INF] $Time $text"
      break
      }
    1 {
      $Global:BackupError = $true
      Write-Host -ForegroundColor Yellow "$text"
      add-content -path $Logfile -value "[WAR] $Time $text"
      break
      }
    2 {
      $Global:BackupError = $true
      Write-Host -ForegroundColor Red "$text"
      add-content -path $Logfile -value "[ERR] $Time $text"
      break
      }
   default {
      Write-Host "$text"
      add-content -path $Logfile -value "[---] $Time $text"
      break
      }
  }
}

#----- Function ----------------------------------------------------------
#        Name: CheckService
# Description: Check the Bentley Select Server Gateway Service
#-------------------------------------------------------------------------
function CheckService
```

```
{
  $Chk = Get-Service -Name $ServiceName
  if ($Chk.Status -ne "Running")
  {
    Log -text " Service not running $ServiceName." -severity 2
    Log -text " Restart the Service $SqlServiceName." -severity 0
    Restart-Service -Name $SqlServiceName
    Start-Sleep -Second 10
    Log -text " Start the Service $ServiceName." -severity 0
    Start-Service -Name $ServiceName
    Start-Sleep -Second 10
    Log -text " Service started" -severity 0
  }
  else
  {
    Log -text " Service is running. Nothing to do." -severity 0
  }
}
while ($true)
{
  CheckService
  Start-Sleep -Second 10
}
```

Listing 3.7: Beispiel Check Service

3.8 Mount Shares, Registry

Mit dem Cmdlet `PSDrive` kann man Shares, Registry und xxx mit einem Namen mounten. Dieser Mountpunkt ist nur für die PowerShell verwendbar und nicht im Windows Explorer zu sehen.

```
PS D:\>Get-PSDrive
Name  Used (GB)     Free (GB)     Provider     Root
----  ---------     ---------     --------     ----
Alias                                 Alias
C 143,39        50,61     FileSystem   C:\
HKCU           Registry     HKEY_CURRENT_USER

PS D:\>Get-PSDrive -Name C
Name     Used (GB)     Free (GB)     Provider     Root
----  ---------     ---------     --------     ----
C 143,39        50,61     FileSystem   C:\

PS D:\>Remove-PSDrive -Name HKCU

PS D:\>cat MountShare.ps1
# Define Shares and Drive Letter
$MyShares = @("\\srv1\daten","\\srv1\public","\\srv1\multimedia")
$MYDrive = "L","P","M"
for ($i=0; $i -lt $MyShares.Length; $i++)
{
 # -Credentials aus den Loesungen PasswortFileAuslesen
 PS D:\>New-PSDrive -Name $MyDrive[$i] -PSProvider FileSystem ‘
 -Root $MyShares[$i] -Credential $MyPasswd -Persist
}
```

Listing 3.8: Beispiel PSDrive

Eine Liste, welche `PSProvider` es für die PowerShell gibt, kann man mit der nachfolgenden Abfrage sich anzegen lassen.

```
D:\>Get-PSProvider
Name                 Capabilities
----                 ------------
WSMan                Credentials
Alias                ShouldProcess
Environment          ShouldProcess
FileSystem           Filter, ShouldProcess
Function             ShouldProcess
Registry             ShouldProcess, Transactions
Variable             ShouldProcess
Certificate          ShouldProcess
```

Listing 3.9: Anzeige PSProvider

Soll ein Share verbunden und im Windows Explorer angezeigt werden, so kann man das entweder mit dem Befehl `net use` machen oder man nimmt `Wscript.Network` für das Verbinden.

```
function Map-Adrive{
<#
  .Example
    Map-Adrive Z \\server\folder
  .Example
    Map-Adrive Z \\server\folder -persistent
  .Example
    Map-Adrive Z \\server\folder -verbose
#>
  [CmdletBinding()]
  param(
```

```
    [string]$driveletter,
    [string]$path,
    [switch]$persistent
  )
  process{
    $nwrk=new-object -com Wscript.Network
    Write-Verbose "Mapping $($driveletter+':') to $path and persist=
       $persistent"
    try{
        #$nwrk.MapNetworkDrive($($driveletter+':'),$path)
        $nwrk.RemoveNetworkDrive($driveletter+':') # remove driveletter
        Write-Verbose "Mapping successful."
       }
   catch{
        Write-Verbose "Mapping failed!"
       }
  }
}
get-help Map-Adrive -example

Map-Adrive B \\dude234\cae_backup
```

Listing 3.10: Beispiel Wscript.Network

In den neueren PowerShell Versionen, ab Version 4, gibt es das Cmdlet `SmbMapping`. Auch mit diesem Cmdlet kann man Shares verbinden.

```
D:\>New-SmbMapping -LocalPath 'M:' -RemotePath '\\srv1\Daten' -Persistent
    $True
Status     Local Path     Remote Path
------     ----------     -----------
OK         M:             \\srv1\daten

D:\>Get-SmbMapping
Status     Local Path     Remote Path
------     ----------     -----------
OK         M:             \\srv1\daten
OK         P:             \\srv1\public
D:\>Remove-SmbMapping M:
```

Listing 3.11: Beispiel SmbMapping

3.9 Last Boot Time

Möchte man angezeigt bekommen, wann der letzte Neustart des Computers war, so kann man das mit einem `Get-WmiObject Win32_OperatingSystem` Objekt machen.

```
PS D:\>$cimString = (Get-WmiObject Win32_OperatingSystem).LastBootUpTime
PS D:\>$BootTime = [Management.ManagementDateTimeConverter]::ToDateTime(
   $cimString)
PS D:\>$BootTime

Freitag, 3. Mai 2019 09:24:39

PS D:\># Ab PowerShell 4
PS D:\>Get-CimInstance -ClassName win32_operatingsystem | select csname,
   lastbootuptime

csname   lastbootuptime
------   --------------
SRV01    04.05.2019 03:42:44
```

Listing 3.12: Last Boot Time

3.10 PopUp

Eine Message Box auf dem Bildschirm ausgeben, kann man mittels eines WSH Objektes machen.

```
PS D:\>cat PopUp.ps1
$StrWshShell=New-Object -ComObject wscript.shell
$StrMsg=$StrWshShell.popup("Hier steht die Message.",0,"Kopfzeile",4)
```

Listing 3.13: Beispiel PopUp

Die erste Zahl gibt die Zeit an, wie lange das PopUp angezeigt werden soll. Wird als Wert eine Null angegeben, so schließt sich das PopUp nicht automatisch, sondern wartet auf eine Aktion. Die letzte Zahl gibt an, welche Buttons angezeigt werden.

Wert	Buttons
0	Ok
1	Ok, Abbrechen
2	Abbrechen, Wiederholen, Ignorieren
3	Ja, Nein, Abbrechen
4	Ja, Nein
5	Wiederholen, Abbrechen
6	Abbrechen, Wiederholen, Weiter

Tabelle 3.1: Buttons

3.11 Dateien anzeigen

Möchte man Dateien in einem Verzeichnis sich anzeigen lassen, so kann man das mit `Get-ChildItem` oder mit den Aliasen `ls, dir` und `gci` machen. Gibt man bei dem Aufruf von Get-ChilItem die Option `-Force` mit an, so werden auch versteckte Dateie angezeigt. Mit der Option `-Recurse` werden auch die Unterverzeichnisse durchsucht. In der PowerShell Version 5 ist ein neuer Schalter hinzugekommen. Mit `-Depth 2` werden in Verbindung mit `-Recurse` nur die Dateien aufgelistet, bis zur 2ten Verzeichnis Ebene sich befinden. Tiefere Dateien und Verzeichnisse werden nicht angezeigt. Es wird also die Suchtiefe angegeben.

```
PS D:\>cat DemoListFiles.ps1
# Listet alle Dateien auf, die den Namen *example* haben
Write-Host -ForegroundColor Gray "Alle Dateien mit *.example*"
gci -Recurse | Where {$_.Name -like "*example*"}

# Alle Dateien mit der Endung ps1
Write-Host -ForegroundColor Gray "`nAlle Dateien mit der Endung *.ps1`n"
gci -Recurse | Where {$_.Extension -eq ".ps1"}

# Anzahl der Dateien mit der Endung .ps1
Write-Host -ForegroundColor Gray "`nAnzahl der .ps1 Dateien inkl.
   Unterordner.`n"
(gci -Recurse).count

Write-Host -ForegroundColor Gray "`nAnzeige mit Filter.`n"
gci -filter *.ps1

Write-Host -ForegroundColor Gray "`nAnzeige Suchkriterium Groesse`n"
gci . -Recurse -Filter *.ps1 | Where {$_.length -gt 1000}

# Mehrere Verzeichnisse auflisten
Write-Host -ForegroundColor Gray "`nMehrere Verzeichnisse anzeigen`n"
gci .\Daten, .\Bilder, .\Texte
```

Listing 3.14: Beispiele Get-ChildItem

Zwei Beispiele, wo nach Extension gesucht wird und das Erstell-Datum mit ausgegeben wird. In dem zweiten Beispiel werden Dateien gelöscht, die älter als n-Tage sind.

```
Try

{

  Write-Host "List all NWD Files in $NavisAppPath"

  Foreach ($File in (Get-Childitem "$NavisAppPath" | Where {$_.Extension -eq
      ".nwd"}))

  {

    $CDate = "{0:dd.MM.yyyy hh:mm:ss}" -f (Get-Childitem -Path "
       $NavisAppPath\$File").CreationTime

    Write-Host "File: $File $CDate"

  }

}

Catch

{
```

```
  Write-Host "Could not found any NWD File in $NavisAppPath"

}
```

Listing 3.15: Beispiel Extension und Date

```
[int]$PurgeDay = 3

Try

{

  Write-Host "Erase all NWD Files older than $PurgeDay day."

  Foreach ($File in (Get-Childitem -Path "$NavisAppPath\*.nwd" | Where-
     Object {$_.CreationTime -lt (Get-Date).AddDays(-$PurgeDay)}))

  {

    Write-Host -ForegroundColor Yellow "Purge File: $File"

  }

}

Catch

{

  Write-Host -ForegroundColor Red "Could not found any NWD File in
     $NavisAppPath"

}
```

Listing 3.16: Beispiel Erase

3.12 Dateien lesen / schreiben

3.12.1 Lesen

Um Dateien einzulesen, gibt es das Cmdlet `Get-Content`. Der Inhalt einer Datei wird in einem Array als String-Objekt gespeichert. Mit dem Parameter `TotalCount` kan man die Anzahl der einzulesenden Zeilen angeben.

```
PS D:\># Ganze Datei einlesen
PS D:\>Get-Content -Path $env:SystemRoot\Win.ini

PS D:\># Nur die erste Zeile einlesen
PS D:\>Get-Content -Path $env:SystemRoor\Win.ini -TotalCount 1

PS D:\> Die vierte Zeile einlesen
PS D:\>(Get-Content -Path $env:SystemRoor\Win.ini)[3]

PS D:\># Die letzte Zeile einlesen
PS D:\>$LastLine = (Get-Content -Path $env:SystemRoor\Win.ini);$LastLine[
   $LastLine.Length-1]

PS D.\># Mit einem negativen Index die letzte Zeile einlesen
PS D:\>(Get-Content -Path $env:SystemRoor\Win.ini)[-1]

PS D:\># Enlesen als ASCII-Datei, jedes Zeichen als 8-Bit-ASCII-Zeichen
PS D:\>Get-Content -Path $env:SystemRoor\Win.ini -Encoding ASCII
```

Listing 3.17: Beispiele Datei lesen

3.13 Read Ini-Datei

Soll eine Ini-Datei eingelesen werden und gleichzeitig Variablen von der Angaben erstellt werden, so kann man das mit dem folgenden Beispiel erledigen. Als Parameter können für die Option -Scope folgende Werte angegeben werden.

- Global - Variablen können in allen PowerShell Prozesse verwendet werden.
- Local - Variablen sind nur in dem aktuellen Bereich gültig.
- Script - Die Variable ist nur in dem Script / Module gültig.
- Private - Die Variable kann nicht außerhalb benutzt werden.
- Eine Nummer - Eine Zahl relativ zum aktuellen Bereich (0 bis Anzahl der Bereiche, wobei 0 der aktuelle Bereich, 1 der übergeordnete Bereich, 2 der übergeordnete Bereich des übergeordneten Bereichs usw. ist). Negative Zahlen können nicht verwendet werden.

```
Param ( $InputFile )

$IniFile = Get-Content $InputFile

$ResultTable=@()
foreach ($line in $IniFile)
{
  Write-Host -ForegroundColor Gray "Processing $line"
  if ($line[0] -eq "#")
  {
    Write-Host -ForegroundColor Green "Skip comment line"
  }
  elseif ($line[0] -eq "[")
  {
    $segment = $line.replace("[","").replace("]","")
    Write-Host -ForegroundColor Gray "Found new segment: $segment"
  }
  elseif ($line -like "*=*")
  {
    Write-Host -ForegroundColor green "Found Keyline"
    # Wenn vor und nach dem = ein Leerzeichen steht.
    $VarName = ($line.split("*=*")[0]).Trim()
    $VarValue = $line.split("*=*")[1]
    #New-Variable -Name $VarName.Substring(0,$VarName.Length-1) -Value
       $VarValue.Substring(1) -force -Scope global
    # Short variante
    New-Variable -Name $VarName -Value $VarValue.Trim() -force -Scope global
    # Funktion ohne Leerzeichen vor und nach dem =
    #New-Variable -Name $line.split("*=*")[0] -Value $line.split("*=*")[1] -
       force -Scope global
    Write-Host -ForegroundColor Gray " New Variable: " $line.split("=")[0] "
       ==> Value: " $line.split("=")[1]
    # oder
    #Write-Host -ForegroundColor Gray " New Variable: " $VarName "==> Value:
        " $VarValue.Trim()
  }
  else
  {
    Write-Host -ForegroundColor Yellow "Skip line"
  }
}
```

Listing 3.18: Read Ini-Datei

3.14 Powershell 32/64bit

Möchte man erfahren, ob die PowerShell Sesion eine 32bit oder 64bit Anwendung ist, so kann man das mit der nachfolgenden Skript machen.

```
$PsBit = [IntPtr]::Size
if ( $PsBit -eq 4 )
{
  Write-Host -ForegroundColor Yellow " PowerShell 32bit running."
}
elseif ( $PsBit -eq 8 )
{
  Write-Host -ForegroundColor Yellow " PowerShell 64bit running."
}
else
{
  Write-Host -ForegroundColor Gray " Undefined $PsBit."
}

# Alternative

switch ([IntPtr]::Size)
{
  4
  {
    Write-Host -ForegroundColor Green " PowerShell 32bit running."
  }
  8
  {
    Write-Host -ForegroundColor Green " PowerShell 64bit running."
  }
  default
  {
    Write-Host -ForegroundColor Gray " Undefined $PsBit."
  }
}
```

Listing 3.19: PowerShell 32/64bit

3.15 Read / Write Streams

Mit dem Windows Dateisystem NTFS ist es Möglich, an einer Datei oder auch Laufwerk ein Stream anzuhängen. Dieser Stream ist erstmal in dem Explorer nicht zu sehen. In dem ersten Beispiel wird nach Streams gesucht.

```
#Grabs the path you wish to search
$getPath = Read-Host "What is the path you would like to search (example: c
    :\temp)"

#recursively searches through a path and grabs the data streams
$getItem = Get-ChildItem -Path $getPath -Recurse | Get-Item -Stream *


foreach($item in $getItem) {

    #outputs the information to the console
    Write-Host "Path: " $item.PSPath -ForegroundColor Green
    Write-Host "Parent Path: " $item.PSParentPath -ForegroundColor Yellow
    Write-Host "PSChildName: " $item.PSChildName -ForegroundColor Yellow
    Write-Host "PSProvider: " $item.PSProvider -ForegroundColor Yellow
    Write-Host "PSIsContainer: " $item.PSIsContainer -ForegroundColor Yellow
    Write-Host "Filename: " $item.FileName -ForegroundColor Yellow
    Write-Host "Stream: " $item.Stream -ForegroundColor Red
    Write-Host "Length: " $item.Length -ForegroundColor Yellow
    Write-Host "`n"

}
```

Listing 3.20: Find Stream

In dem zweiten Beispiel wird ein Stream an einer Datei angehängt."

```
# Example Write and Read alternate data streams (ads)
# Only with PowerShell 3 or higher
$MyFile = "D:\Daten\readme.txt"
Set-Content -Path $MyFile -Value "Hier kÃ¶nnte ihre Werbung stehen"
Get-Content -Path $MyFile

# Add stream: -Stream "StreamName"
Add-Content -Path $MyFile -Value "Password: qwertz" -Stream "secret"
Get-Content -Path $MyFile -Stream "secret"
```

Listing 3.21: Write Stream

3.16 Stopuhr

Um die Laufzeit einer Anwendung zu messen, kann man hierzu das .NET Object `System.Diagnostics.Stopwatch` nehmen.

```
PS D:\># Starten / Initialisierung der Stopuhr
PS D:\>$stopwatch = [system.diagnostics.stopwatch]::StartNew()

PS D:\>$stopwatch.Stop()

PS D:\>$stopwatch.Reset()

PS D:\>$stopwatch.Elapsed

PS D:\>$stopwatch.Elapsed.TotalSeconds
```

Listing 3.22: Stopwatch

Soll die Ausführungszeit eines Befehles gemessen werden, so kann man das mit dem Cmdlet `Measue-Command` machen.

```
PS D:\>[array]$arr
PS D:\>$Perf = Measure-Command -Expression {0..50000 | Foreach-Object { $arr
    += $_ }}

PS D:\>$Perf

PS D:\>$Perf.TotalSeconds
```

Listing 3.23: Measure-Command

Kapitel 4

Verweise

4.0.1 Links

Hier ist eine Auflistung von Links, wo es weitere Informationen über die PowerShell gibt.

URL / Link	Beschreibung
https://www.msxfaq.de/code/powershell/index.htm	PowerShell Allgemein
Windows PowerShell 2.0 Crashkurs	Schnelleinstieg

Tabelle 4.1: Weitere Informationen