Kostenloses eBook

LERNEN PowerShell

Free unaffiliated eBook created from
Stack Overflow contributors.

#powershell

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit PowerShell ... 2

Bemerkungen ... 2

Versionen ... 2

Examples ... 2

Installation oder Setup ... 2

Windows ... 3

Andere Plattformen ... 3

Lassen Sie auf Ihrem Computer gespeicherte Skripts nicht signiert ausgeführt werden ... 3

Aliase und ähnliche Funktionen ... 4

Die Pipeline - Verwenden der Ausgabe eines PowerShell-Cmdlets ... 5

Kommentieren ... 6

.Net-Bibliotheksmethoden aufrufen ... 6

Objekte erstellen ... 7

Kapitel 2: ActiveDirectory-Modul ... 9

Einführung ... 9

Bemerkungen ... 9

Examples ... 9

Modul ... 9

Benutzer ... 9

Gruppen ... 10

Computers ... 10

Objekte ... 10

Kapitel 3: Aliase ... 12

Bemerkungen ... 12

Examples ... 13

Get-Alias ... 13

Set-Alias ... 13

Kapitel 4: Amazon Web Services (AWS) - einfacher Speicherdienst (S3) ... 15

Einführung ... 15

Parameter 15
Examples 15
Erstellen Sie einen neuen S3-Bucket 15
Laden Sie eine lokale Datei in einen S3-Bucket hoch 15
Löschen Sie einen S3-Bucket 16
Kapitel 5: Anonymisieren Sie IP (v4 und v6) in einer Textdatei mit Powershell 17
Einführung 17
Examples 17
IP-Adresse in Textdatei anonymisieren 17
Kapitel 6: Anweisung wechseln 19
Einführung 19
Bemerkungen 19
Examples 19
Einfacher Schalter 19
Wechselanweisung mit Regex-Parameter 19
Einfacher Schalter mit Pause 20
Wechselanweisung mit Platzhalterparameter 20
Wechselanweisung mit genauem Parameter 21
Switch-Anweisung mit CaseSensitive-Parameter 21
Wechselanweisung mit Dateiparameter 22
Einfacher Switch mit Standardbedingung 22
Anweisung mit Ausdrücken wechseln 23
Kapitel 7: Arbeiten mit der PowerShell-Pipeline 24
Einführung 24
Syntax 24
Bemerkungen 24
Examples 25
Schreibfunktionen mit Advanced Lifecycle 25
Grundlegende Pipeline-Unterstützung in Funktionen 25
Arbeitskonzept der Pipeline 26
Kapitel 8: Archivmodul 27
Einführung 27

Syntax ... 27
Parameter ... 27
Bemerkungen ... 28
Examples ... 28
Compress-Archiv mit Platzhalter ... 28
Bestehende ZIP mit Compress-Archiv aktualisieren ... 28
Extrahieren Sie eine Zip mit Expand-Archive ... 28
Kapitel 9: Ausführbare Dateien ausführen ... 29
Examples ... 29
Konsolenanwendungen ... 29
GUI-Anwendungen ... 29
Konsolenströme ... 29
Beendigungscodes ... 30
Kapitel 10: Automatische Variablen ... 31
Einführung ... 31
Syntax ... 31
Examples ... 31
$ pid ... 31
Boolesche Werte ... 31
$ null ... 31
$ OFS ... 32
$ _ / $ PSItem ... 32
$? ... 33
$ Fehler ... 33
Kapitel 11: Automatische Variablen - Teil 2 ... 35
Einführung ... 35
Bemerkungen ... 35
Examples ... 35
$ PSVersionTable ... 35
Kapitel 12: Bedingte Logik ... 36
Syntax ... 36
Bemerkungen ... 36

Examples ... 36
wenn, sonst und sonst wenn ... 36
Negation ... 37
Wenn bedingte Abkürzung ... 37
Kapitel 13: Berechnete Eigenschaften ... 39
Einführung ... 39
Examples ... 39
Dateigröße in KB - Berechnete Eigenschaften anzeigen ... 39
Kapitel 14: Cmdlet-Benennung ... 40
Einführung ... 40
Examples ... 40
Verben ... 40
Substantive ... 40
Kapitel 15: CSV-Analyse ... 41
Examples ... 41
Grundlegende Verwendung von Import-Csv ... 41
Importieren Sie aus CSV- und Umwandlungseigenschaften in den richtigen Typ ... 41
Kapitel 16: Dynamische PowerShell-Parameter ... 43
Examples ... 43
Dynamischer Parameter "einfach" ... 43
Kapitel 17: Einbetten von verwaltetem Code (C # | VB) ... 45
Einführung ... 45
Parameter ... 45
Bemerkungen ... 45
Hinzugefügte Typen werden entfernt ... 45
CSharp- und .NET-Syntax ... 45
Examples ... 46
C # Beispiel ... 46
VB.NET-Beispiel ... 46
Kapitel 18: Einführung in Pester ... 48
Bemerkungen ... 48

Examples ... 48
Erste Schritte mit Pester ... 48
Kapitel 19: Einführung in Psake ... 50
Syntax ... 50
Bemerkungen ... 50
Examples ... 50
Grundriss ... 50
Beispiel für FormatTaskName ... 50
Task bedingt ausführen ... 51
ContinueOnError ... 51
Kapitel 20: Eingebaute Variablen ... 52
Einführung ... 52
Examples ... 52
$ PSScriptRoot ... 52
$ Args ... 52
$ PSItem ... 52
$? ... 53
$ Fehler ... 53
Kapitel 21: Email schicken ... 54
Einführung ... 54
Parameter ... 54
Examples ... 55
Einfache Send-MailMessage ... 55
Send-MailMessage mit vordefinierten Parametern ... 55
SMTPClient - E-Mail mit Textdatei im Nachrichtentext ... 56
Kapitel 22: Erkennung von Amazon Web Services (AWS) ... 57
Einführung ... 57
Examples ... 57
Erkennen von Bildbeschriftungen mit AWS Recognition ... 57
Vergleichen Sie die Gesichtsähnlichkeit mit AWS Rekognition ... 58
Kapitel 23: Erstellen von klassenbasierten DSC-Ressourcen ... 59
Einführung ... 59

Bemerkungen 59
Examples 59
Erstellen Sie eine DSC-Ressource-Skeleton-Klasse 59
DSC-Ressourcenskelett mit Schlüsseleigenschaft 59
DSC-Ressource mit obligatorischer Eigenschaft 60
DSC-Ressource mit erforderlichen Methoden 60
Kapitel 24: Fehlerbehandlung 62
Einführung 62
Examples 62
Fehlertypen 62
Kapitel 25: Fortschrittsbalken verwenden 64
Einführung 64
Examples 64
Einfache Verwendung der Fortschrittsleiste 64
Verwendung des inneren Fortschrittsbalkens 65
Kapitel 26: Gemeinsame Parameter 67
Bemerkungen 67
Examples 67
ErrorAction-Parameter 67
-ErrorAction Weiter 67
-ErrorAction Ignorieren 68
-ErrorAction-Anfrage 68
-ErrorAction SilentlyContinue 68
-ErrorAction Stop 68
-ErrorAction Suspend 69
Kapitel 27: Geplantes Aufgabenmodul 70
Einführung 70
Examples 70
Führen Sie das PowerShell-Skript als geplante Aufgabe aus 70
Kapitel 28: Gewünschter Zustand Konfiguration 71
Examples 71

Einfaches Beispiel - Aktivieren von WindowsFeature 71
Starten von DSC (mof) auf einem Remote-Computer 71
Importieren von psd1 (Datendatei) in lokale Variable 71
Liste der verfügbaren DSC-Ressourcen 72
Ressourcen für die Verwendung in DSC importieren 72
Kapitel 29: Grundlegende Vorgänge 73
Einführung 73
Syntax 73
Examples 73
Filtern: Where-Object / Where /? 73
Sortierung: Sortierobjekt / Sortierung 74
Gruppierung: Gruppenobjekt / Gruppe 75
Projizieren: Objekt auswählen / auswählen 75
Kapitel 30: GUI in Powershell 78
Examples 78
WPF-GUI für das Get-Service-Cmdlet 78
Kapitel 31: HashTables 80
Einführung 80
Bemerkungen 80
Examples 80
Hash-Tabelle erstellen 80
Greifen Sie mit einem Schlüssel auf einen Hashtabellenwert zu 80
Durchlaufen einer Hashtabelle 81
Fügen Sie einer vorhandenen Hashtabelle ein Schlüsselwertpaar hinzu 81
Aufzählung durch Schlüssel und Schlüsselwertpaare 82
Entfernen Sie ein Schlüsselwertpaar aus einer vorhandenen Hashtabelle 82
Kapitel 32: Infrastrukturautomatisierung 83
Einführung 83
Examples 83
Einfaches Skript für den Black-Box-Integrationstest von Konsolenanwendungen 83
Kapitel 33: ISE-Modul 84
Einführung 84

Examples ... 84
Skripte testen ... 84
Kapitel 34: Kommentarbasierte Hilfe ... 85
Einführung ... 85
Examples ... 85
Funktionskommentare-basierte Hilfe ... 85
Kommentar-basierte Hilfe ... 87
Kapitel 35: Kommunikation mit RESTful-APIs ... 90
Einführung ... 90
Examples ... 90
Verwenden Sie Slack.com Incoming Webhooks ... 90
Nachricht an hipChat senden ... 90
Verwenden von REST mit PowerShell-Objekten zum Abrufen und Übertragen einzelner Daten ... 90
Verwenden von REST mit PowerShell-Objekten zum Abrufen und POST vieler Elemente ... 91
Verwenden von REST mit PowerShell zum Löschen von Elementen ... 91
Kapitel 36: Mit Objekten arbeiten ... 92
Examples ... 92
Objekte aktualisieren ... 92
Eigenschaften hinzufügen ... 92
Eigenschaften entfernen ... 92
Ein neues Objekt erstellen ... 93
Option 1: Neues Objekt ... 93
Option 2: Auswahlobjekt ... 93
Option 3: pscustomobject-Beschleuniger (PSv3 + erforderlich) ... 94
Objekt untersuchen ... 94
Instanzen generischer Klassen erstellen ... 95
Kapitel 37: Mit XML-Dateien arbeiten ... 97
Examples ... 97
Zugriff auf eine XML-Datei ... 97
XML-Dokument mit XmlWriter () erstellen ... 99
XML-Snippits zum aktuellen XMLDocument hinzufügen ... 100

Beispieldaten ... 100
XML-Dokument ... 100
Neue Daten ... 101
Vorlagen ... 102
Neue Daten hinzufügen ... 102
Profitieren ... 104
Verbesserungen ... 104
Kapitel 38: Module, Skripte und Funktionen ... 105
Einführung ... 105
Examples ... 105
Funktion ... 105
Demo ... 105
Skript ... 106
Demo ... 106
Modul ... 107
Demo ... 107
Erweiterte Funktionen ... 107
Kapitel 39: MongoDB ... 111
Bemerkungen ... 111
Examples ... 111
MongoDB mit C # -Treiber 1.7 mit PowerShell ... 111
Ich habe 3 Sätze von Arrays in Powershell ... 111
Kapitel 40: Operatoren ... 113
Einführung ... 113
Examples ... 113
Rechenzeichen ... 113
Logische Operatoren ... 113
Zuweisungsoperatoren ... 113
Vergleichsoperatoren ... 114
Umleitungsoperatoren ... 114
Mischen von Operandentypen: Der Typ des linken Operanden bestimmt das Verhalten ... 115

String-Manipulationsoperatoren ... 116
Kapitel 41: Paketverwaltung ... 117
Einführung ... 117
Examples ... 117
Suchen Sie ein PowerShell-Modul anhand eines Musters ... 117
Erstellen Sie das Standard-Reposity-PowerShell-Modul ... 117
Suchen Sie ein Modul anhand des Namens ... 117
Installieren Sie ein Modul anhand des Namens ... 117
Deinstallieren Sie ein Modul, meinen Namen und meine Version ... 117
Aktualisieren Sie ein Modul anhand des Namens ... 117
Kapitel 42: Parametersätze ... 118
Einführung ... 118
Examples ... 118
Einfache Parametersätze ... 118
Parametersatz, um die Verwendung eines Parmeters zu erzwingen, wenn ein anderer ausgewählt ... 118
Parametersatz zur Begrenzung der Kombination von Parmetern ... 119
Kapitel 43: PowerShell "Streams"; Debuggen, ausführlich, Warnung, Fehler, Ausgabe und Info 120
Bemerkungen ... 120
Examples ... 120
Write-Output ... 120
Schreibe Einstellungen ... 120
Kapitel 44: Powershell SQL-Abfragen ... 122
Einführung ... 122
Parameter ... 122
Bemerkungen ... 122
Examples ... 124
SQLEexample ... 124
SQLQuery ... 124
Kapitel 45: PowerShell.exe Befehlszeile ... 126
Parameter ... 126
Examples ... 127
Befehl ausführen ... 127

-Command <string> 127
-Befehl {Skriptblock} 127
-Befehl - (Standardeingabe) 127
Skriptdatei ausführen 128
Grundlegendes Skript 128
Verwenden von Parametern und Argumenten 128
Kapitel 46: PowerShell-Funktionen 129
Einführung 129
Examples 129
Einfache Funktion ohne Parameter 129
Grundparameter 129
Pflichtparameter 130
Erweiterte Funktion 131
Parameterüberprüfung 132
ValidateSet 132
ValidateRange 133
ValidatePattern 133
ValidateLength 133
ValidateCount 133
ValidateScript 134
Kapitel 47: PowerShell-Hintergrundjobs 135
Einführung 135
Bemerkungen 135
Examples 135
Grundlegende Schaffung von Arbeitsplätzen 135
Grundlegende Jobverwaltung 136
Kapitel 48: PowerShell-Klassen 138
Einführung 138
Examples 138
Methoden und Eigenschaften 138
Verfügbare Konstruktoren für eine Klasse auflisten 138

Konstruktor überladen ... 140
Holen Sie sich alle Mitglieder einer Instanz ... 140
Grundlegende Klassenvorlage ... 140
Vererbung von übergeordneter Klasse zu untergeordneter Klasse ... 141
Kapitel 49: Powershell-Module ... 142
Einführung ... 142
Examples ... 142
Erstellen Sie ein Modulmanifest ... 142
Einfaches Modulbeispiel ... 142
Eine Variable aus einem Modul exportieren ... 143
Strukturieren von PowerShell-Modulen ... 143
Lage der Module ... 144
Sichtbarkeit von Modulmitgliedern ... 144
Kapitel 50: Powershell-Profile ... 145
Bemerkungen ... 145
Examples ... 146
Erstellen Sie ein Basisprofil ... 146
Kapitel 51: Powershell-Remoting ... 147
Bemerkungen ... 147
Examples ... 147
Aktivieren von PowerShell-Remoting ... 147
Nur für Umgebungen außerhalb von Domänen ... 147
Aktivieren der Standardauthentifizierung ... 148
Verbindung zu einem Remote-Server über PowerShell ... 148
Befehle auf einem Remote-Computer ausführen ... 148
Serialisierungswarnung für Remoting ... 149
Argumentverwendung ... 150
Eine bewährte Methode zum automatischen Aufräumen von PSSessions ... 151
Kapitel 52: PowerShell-Workflows ... 152
Einführung ... 152
Bemerkungen ... 152

Examples ... 152
Einfaches Workflow-Beispiel ... 152
Workflow mit Eingabeparametern ... 152
Führen Sie den Workflow als Hintergrundjob aus ... 153
Fügen Sie einem Workflow einen Parallelblock hinzu ... 153
Kapitel 53: PSScriptAnalyzer - PowerShell Script Analyzer ... 154
Einführung ... 154
Syntax ... 154
Examples ... 154
Analysieren von Skripts mit den integrierten voreingestellten Regelsätzen ... 154
Analysieren von Skripts anhand jeder integrierten Regel ... 155
Listet alle integrierten Regeln auf ... 155
Kapitel 54: Regeln der Namensgebung ... 156
Examples ... 156
Funktionen ... 156
Kapitel 55: Reguläre Ausdrücke ... 157
Syntax ... 157
Examples ... 157
Einziges Paar ... 157
Verwenden des -Match-Operators ... 157
Select-String verwenden ... 158
Verwendung von [RegEx] :: Match () ... 159
Ersetzen ... 159
Verwenden Sie -Replace Operator ... 160
Verwendung der Methode [RegEx] :: Replace () ... 160
Ersetzen Sie Text durch einen dynamischen Wert mit einem MatchEvalutor ... 160
Sonderzeichen entkommen ... 161
Mehrere Übereinstimmungen ... 162
Select-String verwenden ... 162
Verwendung von [RegEx] :: Matches () ... 163
Kapitel 56: Schleifen ... 164

Einführung ... 164
Syntax ... 164
Bemerkungen ... 164
Für jeden ... 164
Performance ... 165
Examples ... 165
Zum ... 165
Für jeden ... 165
Während ... 166
ForEach-Objekt ... 167
Grundlegende Verwendung ... 167
Erweiterte Nutzung ... 167
Tun ... 168
ForEach () - Methode ... 168
Fortsetzen ... 169
Brechen ... 169
Kapitel 57: SharePoint-Modul ... 171
Examples ... 171
Laden des SharePoint-Snap-Ins ... 171
Durchlaufen aller Listen einer Websitesammlung ... 171
Rufen Sie alle installierten Funktionen einer Websitesammlung ab ... 171
Kapitel 58: ShouldProcess verwenden ... 173
Syntax ... 173
Parameter ... 173
Bemerkungen ... 173
Examples ... 173
Hinzufügen von -WhatIf und -Confirm Support zum Cmdlet ... 173
Verwendung von ShouldProcess () mit einem Argument ... 173
Full Usage Beispiel ... 174
Kapitel 59: Sicherheit und Kryptographie ... 176
Examples ... 176
Berechnen der Hashcodes eines Strings über .Net Cryptography ... 176

Kapitel 60: Skripte signieren ... 177
Bemerkungen ... 177
Ausführungsrichtlinien ... 177
Examples ... 178
Skript signieren ... 178
Ändern der Ausführungsrichtlinie mit Set-ExecutionPolicy ... 178
Umgehung der Ausführungsrichtlinie für ein einzelnes Skript ... 179
Andere Ausführungsrichtlinien: ... 179
Rufen Sie die aktuelle Ausführungsrichtlinie ab ... 180
Signatur von einem signierten Skript abrufen ... 180
Erstellen eines selbstsignierten Codesignaturzertifikats zum Testen ... 180
Kapitel 61: Skriptvoraussetzungen erzwingen ... 182
Syntax ... 182
Bemerkungen ... 182
Examples ... 182
Erzwingen Sie die Mindestversion des Powershell-Hosts ... 182
Erzwingen Sie die Ausführung des Skripts als Administrator ... 182
Kapitel 62: Spezielle Bediener ... 184
Examples ... 184
Array-Ausdrucksoperator ... 184
Operation aufrufen ... 184
Punktbeschaffungsoperator ... 184
Kapitel 63: Spritzen ... 185
Einführung ... 185
Bemerkungen ... 185
Examples ... 185
Splatting-Parameter ... 185
Übergeben eines Switch-Parameters mithilfe von Splatting ... 186
Pfeifen und Spritzen ... 186
Splatting von der Top-Level-Funktion zu einer Reihe von inneren Funktionen ... 186
Kapitel 64: TCP-Kommunikation mit PowerShell ... 188

Examples ... 188
TCP-Listener ... 188
TCP Sender ... 188
Kapitel 65: Umgang mit Geheimnissen und Anmeldeinformationen ... 190
Einführung ... 190
Examples ... 190
Aufforderung zur Eingabe von Anmeldeinformationen ... 190
Zugriff auf das Klartext-Passwort ... 190
Mit gespeicherten Anmeldeinformationen arbeiten ... 191
Encrypter ... 191
Der Code, der die gespeicherten Anmeldeinformationen verwendet: ... 191
Speichern der Anmeldeinformationen in verschlüsselter Form und Übergeben der Parameter als ... 192
Kapitel 66: Umgebungsvariablen ... 193
Examples ... 193
Windows-Umgebungsvariablen sind als PS-Laufwerk mit dem Namen Env sichtbar: ... 193
Sofortiger Aufruf von Umgebungsvariablen mit $ env: ... 193
Kapitel 67: URL-Kodierung / Dekodierung ... 194
Bemerkungen ... 194
Examples ... 194
Schnellstart: Kodierung ... 194
Schnellstart: Dekodierung ... 195
Encode Query String mit [uri] :: EscapeDataString () ` ... 195
Encode-Abfragezeichenfolge mit [System.Web.HttpUtility] :: UrlEncode () ` ... 196
URL mit `[uri] :: UnescapeDataString ()` decodieren ... 196
URL mit `[System.Web.HttpUtility] :: UrlDecode ()` decodieren ... 198
Kapitel 68: Variablen in PowerShell ... 201
Einführung ... 201
Examples ... 201
Einfache Variable ... 201
Eine Variable entfernen ... 201
Umfang ... 201
Lesen einer CmdLet-Ausgabe ... 202

Listenzuordnung mehrerer Variablen ... 203
Arrays ... 204
Hinzufügen zu einer Arry ... 204
Arrays miteinander kombinieren ... 204
Kapitel 69: Verhalten in PowerShell zurückgeben ... 205
Einführung ... 205
Bemerkungen ... 205
Examples ... 205
Vorzeitige Ausfahrt ... 205
Erwischt! Rückkehr in die Pipeline ... 206
Erwischt! Unerwünschte Ausgabe ignorieren ... 206
Rückgabe mit einem Wert ... 206
Wie man mit Funktionen arbeitet, kehrt zurück ... 207
Kapitel 70: Verwenden des Hilfesystems ... 209
Bemerkungen ... 209
Examples ... 209
Aktualisieren des Hilfesystems ... 209
Get-Help verwenden ... 209
Online-Version eines Hilfethemas anzeigen ... 210
Beispiele ansehen ... 210
Anzeigen der vollständigen Hilfeseite ... 210
Anzeigen der Hilfe für einen bestimmten Parameter ... 210
Kapitel 71: Vorhandene statische Klassen verwenden ... 211
Einführung ... 211
Examples ... 211
Erstellen Sie sofort eine neue GUID ... 211
Verwenden der .Net Math-Klasse ... 211
Typen hinzufügen ... 212
Kapitel 72: Wie kann ich das neueste Artefakt von Artifactory mithilfe des Powershell-Skri ... 213
Einführung ... 213
Examples ... 213
Powershell-Skript zum Herunterladen des neuesten artifcat ... 213

Kapitel 73: WMI und CIM ... 214
Bemerkungen ... 214
CIM vs. WMI ... 214
Zusätzliche Ressourcen ... 215
Examples ... 215
Objekte abfragen ... 215
Alle Objekte für die CIM-Klasse auflisten ... 215
Filter verwenden ... 215
Verwendung einer WQL-Abfrage: ... 216
Klassen und Namespaces ... 217
Liste der verfügbaren Klassen ... 217
Suche nach einer Klasse ... 218
Listen Sie Klassen in einem anderen Namespace auf ... 218
Liste der verfügbaren Namespaces ... 219
Kapitel 74: Zeichenketten ... 221
Syntax ... 221
Bemerkungen ... 221
Examples ... 221
Erstellen einer Basiszeichenfolge ... 221
String ... 221
Wörtliche Zeichenfolge ... 221
Zeichenfolge formatieren ... 222
Mehrzeilige Zeichenfolge ... 222
Here-String ... 222
Here-String ... 222
Wörtliche Here-Zeichenfolge ... 223
Zeichenketten verketten ... 223
Verwenden von Variablen in einer Zeichenfolge ... 223
Verwenden Sie den Operator + ... 223
Unterausdrücke verwenden ... 224

Spezielle Charaktere 224

Credits **225**

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: powershell

It is an unofficial and free PowerShell ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PowerShell.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit PowerShell

Bemerkungen

Windows PowerShell ist eine Shell- und Skriptkomponente des Windows Management Framework, eines Frameworks zur Automatisierung / Konfiguration von Microsoft, das auf .NET Framework basiert. PowerShell wird standardmäßig auf allen unterstützten Versionen von Windows-Client- und -Serverbetriebssystemen seit Windows 7 / Windows Server 2008 R2 installiert. Powershell kann jederzeit durch Herunterladen einer späteren Version von Windows Management Framework (WMF) aktualisiert werden. Die "Alpha" -Version von PowerShell 6 ist plattformübergreifend (Windows, Linux und OS X) und muss von dieser Versionsseite heruntergeladen und installiert werden.

Zusätzliche Ressourcen:

- MSDN-Dokumentation: https://msdn.microsoft.com/de-de/powershell/scripting/powershell-scripting
- TechNet: https://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx
    - Über Seiten
- PowerShell Gallery: https://www.powershellgallery.com/
- MSDN-Blog: https://blogs.msdn.microsoft.com/powershell/
- Github: https://github.com/powershell
- Community Site: http://powershell.com/cs/

Versionen

Ausführung	In Windows enthalten	Anmerkungen	Veröffentlichungsdatum
1,0	XP / Server 2008		2006-11-01
2,0	7 / Server 2008 R2		2009-11-01
3,0	8 / Server 2012		2012-08-01
4,0	8.1 / Server 2012 R2		2013-11-01
5,0	10 / Server 2016 Tech Preview		2015-12-16
5.1	10 Jubiläumsausgabe / Server 2016		2017-01-27

Examples

Installation oder Setup

Windows

PowerShell ist im Windows Management Framework enthalten. Installation und Setup sind in modernen Windows-Versionen nicht erforderlich.

Aktualisierungen an PowerShell können durch Installieren einer neueren Version von Windows Management Framework durchgeführt werden.

Andere Plattformen

Die "Beta" -Version von PowerShell 6 kann auf anderen Plattformen installiert werden. Die Installationspakete sind hier verfügbar.

Beispielsweise wird PowerShell 6 für Ubuntu 16.04 veröffentlicht, um Repositorys für eine einfache Installation (und Updates) zu verpacken.

Um zu installieren, führe folgendes aus:

```
# Import the public repository GPG keys
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Register the Microsoft Ubuntu repository
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Update apt-get
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
powershell
```

Nachdem Sie das Microsoft-Repository einmal als Superuser registriert haben, müssen Sie nur noch `sudo apt-get upgrade powershell` , um es zu aktualisieren. Dann einfach `powershell` laufen `powershell`

Lassen Sie auf Ihrem Computer gespeicherte Skripts nicht signiert ausgeführt werden

Aus Sicherheitsgründen ist PowerShell standardmäßig so eingerichtet, dass nur signierte Skripts ausgeführt werden können. Wenn Sie den folgenden Befehl ausführen, können Sie unsignierte Skripts ausführen (dazu müssen Sie PowerShell als Administrator ausführen).

```
Set-ExecutionPolicy RemoteSigned
```

Eine andere Möglichkeit, PowerShell-Skripts auszuführen, ist die Verwendung von `Bypass` als `ExecutionPolicy` :

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

Oder von Ihrer vorhandenen PowerShell-Konsole oder ISE-Sitzung aus, indem Sie Folgendes ausführen:

```
 Set-ExecutionPolicy Bypass Process
```

Eine temporäre Problemumgehung für die Ausführungsrichtlinie kann auch erreicht werden, indem die ausführbare Powershell-Datei ausgeführt wird und eine gültige Richtlinie als Parameter `-ExecutionPolicy` wird. Die Richtlinie ist nur während der Prozessdauer gültig, sodass kein Administratorzugriff auf die Registrierung erforderlich ist.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

Es stehen mehrere andere Richtlinien zur Verfügung, und Websites empfehlen oft die Verwendung von `Set-ExecutionPolicy Unrestricted` . Diese Richtlinie bleibt gültig, bis sie geändert wird, und verringert die Sicherheit des Systems. Dies ist nicht ratsam. Die Verwendung von `RemoteSigned` wird empfohlen, da lokal gespeicherter und geschriebener Code möglich ist und der `RemoteSigned` Code mit einem Zertifikat eines vertrauenswürdigen `RemoteSigned` signiert werden muss.

Beachten Sie auch, dass die Ausführungsrichtlinie möglicherweise durch Gruppenrichtlinien erzwungen wird, sodass Gruppenrichtlinien diese Einstellung möglicherweise selbst dann ändern, wenn sie in systemweit `Unrestricted` geändert wird (in der Regel 15 Minuten). Sie können die Ausführungsrichtlinie sehen, die in den verschiedenen Bereichen mithilfe von `Get-ExecutionPolicy -List`

TechNet-Dokumentation:
Set-ExecutionPolicy
about_Execution_Policies

Aliase und ähnliche Funktionen

In PowerShell gibt es viele Möglichkeiten, um dasselbe Ergebnis zu erzielen. Dies lässt sich gut mit dem einfachen und bekannten `Hello World` Beispiel veranschaulichen:

`Write-Host` :

```
Write-Host "Hello World"
```

Mit `Write-Output` :

```
Write-Output 'Hello world'
```

Es ist erwähnenswert, dass, obwohl `Write-Output` und `Write-Host` beide auf den Bildschirm schreiben, ein geringfügiger Unterschied besteht. `Write-Host` schreibt *nur* in stdout (dh den Konsolenbildschirm), wohingegen `Write-Output` sowohl in stdout *AND als auch* in den output

[success] `Write-Output` schreibt, um eine Umleitung zu ermöglichen . Die Umleitung (und Streams im Allgemeinen) ermöglicht die Ausgabe eines Befehls als Eingabe an einen anderen Befehl, einschließlich der Zuordnung zu einer Variablen.

```
> $message = Write-Output "Hello World"
> $message
"Hello World"
```

Diese ähnlichen Funktionen sind keine Aliasnamen, können jedoch zu den gleichen Ergebnissen führen, wenn der Erfolgsstrom nicht "verschmutzt" werden soll.

`Write-Output` wird auf `Echo` oder `Write`

```
Echo 'Hello world'
Write 'Hello world'
```

Oder indem Sie einfach "Hallo Welt" eingeben!

```
'Hello world'
```

All dies führt zu der erwarteten Konsolenausgabe

```
Hello world
```

Ein anderes Beispiel für Aliasnamen in PowerShell ist die übliche Zuordnung sowohl älterer Eingabeaufforderungsbefehle als auch BASH-Befehle zu PowerShell-Cmdlets. Alle folgenden Elemente erzeugen eine Verzeichnisliste des aktuellen Verzeichnisses.

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

Schließlich können Sie mit dem Cmdlet Set-Alias einen eigenen Alias erstellen. Als Beispiel wollen wir alisas `Test-NetConnection` , die im Wesentlichen die PowerShell ist, die dem Ping-Befehl der Eingabeaufforderung entspricht, als "Ping" bezeichnen.

```
Set-Alias -Name ping -Value Test-NetConnection
```

Jetzt können Sie `ping` anstelle von `Test-NetConnection` ! Beachten Sie, dass Sie die Zuordnung überschreiben, wenn der Alias bereits verwendet wird.

Der Alias bleibt aktiv, bis die Sitzung aktiv ist. Wenn Sie die Sitzung schließen und versuchen, den Alias auszuführen, den Sie in Ihrer letzten Sitzung erstellt haben, funktioniert er nicht. Um dieses Problem zu beheben, können Sie alle Aliase aus einer Excel-Datei einmal in Ihre Sitzung importieren, bevor Sie mit der Arbeit beginnen.

Die Pipeline - Verwenden der Ausgabe eines PowerShell-Cmdlets

Eine der ersten Fragen, die Benutzer bei der Verwendung von PowerShell für die Skripterstellung haben, ist das Ändern der Ausgabe eines Cmdlets, um eine andere Aktion auszuführen.

Das Pipeline-Symbol `|` wird am Ende eines Cmdlets verwendet, um die exportierten Daten zu übernehmen und dem nächsten Cmdlet zuzuführen. Ein einfaches Beispiel ist die Verwendung von Select-Object, um nur die Name-Eigenschaft einer Datei anzuzeigen, die in Get-ChildItem angezeigt wird:

```
Get-ChildItem | Select-Object Name
#This may be shortened to:
gci | Select Name
```

Durch die erweiterte Nutzung der Pipeline können wir die Ausgabe eines Cmdlets in eine foreach-Schleife leiten:

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#This may be shortened to:
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Beachten Sie, dass das obige Beispiel die automatische Variable $ _ verwendet. $ _ ist der kurze Alias von $ PSItem, einer automatischen Variablen, die das aktuelle Element in der Pipeline enthält.

Kommentieren

Um Kommentare zu Energieskripten zu kommentieren, indem Sie die Zeile mit dem `#` (Hash) -Symbol voranstellen

```
# This is a comment in powershell
Get-ChildItem
```

Sie können auch mehrzeilige Kommentare mit `<#` und `#>` am Anfang und Ende des Kommentars einfügen.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

.Net-Bibliotheksmethoden aufrufen

Statische .Net-Bibliotheksmethoden können von PowerShell aus aufgerufen werden, indem der vollständige Klassenname in der dritten Klammer gekapselt wird und die Methode dann aufgerufen wird `::`

```
#calling Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

Statische Methoden können von der Klasse selbst aufgerufen werden. Für den Aufruf nicht statischer Methoden ist jedoch eine Instanz der .Net-Klasse (ein Objekt) erforderlich.

Beispielsweise kann die AddHours-Methode nicht von der System.DateTime-Klasse selbst aufgerufen werden. Es erfordert eine Instanz der Klasse:

```
C:\> [System.DateTime]::AddHours(15)
Method invocation failed because [System.DateTime] does not contain a method named 'AddHours'.
At line:1 char:1
+ [System.DateTime]::AddHours(15)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

In diesem Fall erstellen wir zuerst ein Objekt , zum Beispiel:

```
C:\> $Object = [System.DateTime]::Now
```

Dann können wir Methoden dieses Objekts verwenden, auch Methoden, die nicht direkt von der System.DateTime-Klasse aufgerufen werden können, wie die AddHours-Methode:

```
C:\> $Object.AddHours(15)

Monday 12 September 2016 01:51:19
```

Objekte erstellen

Das Cmdlet `New-Object` wird zum Erstellen eines Objekts verwendet.

```
# Create a DateTime object and stores the object in variable "$var"
$var = New-Object System.DateTime

# calling constructor with parameters
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

In vielen Fällen wird ein neues Objekt erstellt, um Daten zu exportieren oder an ein anderes Commandlet zu übergeben. Dies kann wie folgt gemacht werden:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
    Role = "Interface"
    Environment = "Production"
}
```

Es gibt viele Möglichkeiten, ein Objekt zu erstellen. Die folgende Methode ist wahrscheinlich die kürzeste und schnellste Möglichkeit, ein `PSCustomObject` zu erstellen:

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role         = 'Interface'
    Environment  = 'Production'
}
```

Wenn Sie bereits über ein Objekt verfügen, aber nur eine oder zwei zusätzliche Eigenschaften benötigen, können Sie diese Eigenschaft einfach mit `Select-Object` hinzufügen:

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
    @{Name='PropertieName'; Expression={'CustomValue'}}
```

Alle Objekte können in Variablen gespeichert oder in die Pipeline übergeben werden. Sie können diese Objekte auch zu einer Sammlung hinzufügen und dann die Ergebnisse am Ende anzeigen.

Sammlungen von Objekten funktionieren gut mit Export-CSV (und Import-CSV). Jede Zeile der CSV ist ein Objekt, jede Spalte eine Eigenschaft.

Formatierungsbefehle konvertieren Objekte in einen Textstrom zur Anzeige. Vermeiden Sie die Verwendung von Format- * -Befehlen bis zum letzten Schritt der Datenverarbeitung, um die Verwendbarkeit der Objekte zu erhalten.

Erste Schritte mit PowerShell online lesen: https://riptutorial.com/de/powershell/topic/822/erste-schritte-mit-powershell

Kapitel 2: ActiveDirectory-Modul

Einführung

In diesem Thema werden einige grundlegende Cmdlets vorgestellt, die im Active Directory-Modul für PowerShell zum Bearbeiten von Benutzern, Gruppen, Computern und Objekten verwendet werden.

Bemerkungen

Denken Sie daran, dass das Hilfesystem von PowerShell eine der besten Ressourcen ist, die Sie möglicherweise verwenden können.

```
Get-Help Get-ADUser -Full
Get-Help Get-ADGroup -Full
Get-Help Get-ADComputer -Full
Get-Help Get-ADObject -Full
```

In der gesamten Hilfedokumentation finden Sie Beispiele, Syntax und Parameterhilfe.

Examples

Modul

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

Benutzer

Rufen Sie den Active Directory-Benutzer ab

```
Get-ADUser -Identity JohnSmith
```

Rufen Sie alle mit dem Benutzer verknüpften Eigenschaften ab

```
Get-ADUser -Identity JohnSmith -Properties *
```

Ausgewählte Eigenschaften für Benutzer abrufen

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name,
Mail
```

Neuer AD-Benutzer

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path
"CN=Users,DC=Domain,DC=Local"
```

Gruppen

Rufen Sie die Active Directory-Gruppe ab

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Rufen Sie alle mit der Gruppe verknüpften Eigenschaften ab

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Alle Mitglieder einer Gruppe abrufen

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Hinzufügen eines AD-Benutzers zu einer AD-Gruppe

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Neue AD-Gruppe

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

Computers

AD-Computer abrufen

```
Get-ADComputer -Identity "JohnLaptop"
```

Rufen Sie alle mit dem Computer verknüpften Eigenschaften ab

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Wählen Sie die Eigenschaften des Computers aus

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Objekte

Rufen Sie ein Active Directory-Objekt ab

```
#Identity can be ObjectGUID, Distinguished Name or many more
Get-ADObject -Identity "ObjectGUID07898"
```

Verschieben Sie ein Active Directory-Objekt

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath
"OU=SuperUser,DC=Domain,DC=Local"
```

Ändern Sie ein Active Directory-Objekt

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is
My First Object Modification"
```

ActiveDirectory-Modul online lesen:
https://riptutorial.com/de/powershell/topic/8213/activedirectory-modul

Kapitel 3: Aliase

Bemerkungen

Das Powershell-Benennungssystem hat recht strenge Regeln für die Benennung von Cmdlets (Verb-Noun-Vorlage; weitere Informationen finden Sie in [Thema wurde noch nicht erstellt]). Es ist jedoch nicht wirklich praktisch, `Get-ChildItems` jedes Mal zu schreiben, wenn Sie Dateien interaktiv in einem Verzeichnis `Get-ChildItems` möchten.
Daher ermöglicht Powershell die Verwendung von Abkürzungen - Aliasnamen - anstelle von Cmdlet-Namen.

Sie können `ls` , `dir` oder `gci` anstelle von `Get-ChildItem` schreiben und erhalten dasselbe Ergebnis. Alias entspricht seinem Cmdlet.

Einige der häufigsten Aliase sind:

alias	Cmdlet
%, für jeden	Für-EachObject
?, woher	Wo-Objekt
Katze, gc, Typ	Get-Content
cd, chdir, sl	Ort festlegen
cls, klar	Clear-Host
cp, kopie, cpi	Artikel kopieren
dir / ls / gci	Get-ChildItem
Echo, schreibe	Write-Output
fl	Format-Liste
ft	Format-Tabelle
fw	Format breit
gc, pwd	Get-Location
gm	Get-Member
iex	Aufruf-Ausdruck
ii	Invoke-Item

alias	Cmdlet
mv, umziehen	Artikel verschieben
rm, rmdir, del, löschen, rd, ri	Gegenstand entfernen
schlafen	Start-Schlaf
Start, Saps	Startvorgang

In der Tabelle oben können Sie sehen, wie Aliase das Simulieren von Befehlen aus anderen Umgebungen (cmd, bash) ermöglichten, wodurch die Erkennbarkeit erhöht wurde.

Examples

Get-Alias

So listen Sie alle Aliase und ihre Funktionen auf:

```
Get-Alias
```

So erhalten Sie alle Aliase für ein bestimmtes Cmdlet:

```
PS C:\> get-alias -Definition Get-ChildItem

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

So suchen Sie nach Aliasnamen:

```
PS C:\> get-alias -Name p*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           popd -> Pop-Location
Alias           proc -> Get-Process
Alias           ps -> Get-Process
Alias           pushd -> Push-Location
Alias           pwd -> Get-Location
```

Set-Alias

Mit diesem Cmdlet können Sie neue alternative Namen zum Beenden von Cmdlets erstellen

```
PS C:\> Set-Alias -Name proc -Value Get-Process
PS C:\> proc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName
```

```
-------  ------    -----      ----- -----   ------     --  -- -----------
    292      17    13052      20444 ...19     7.94    620   1 ApplicationFrameHost
....
```

Beachten Sie, dass alle von Ihnen erstellten Aliasnamen nur in der aktuellen Sitzung bestehen bleiben. Wenn Sie eine neue Sitzung starten, müssen Sie Ihre Aliase erneut erstellen. PowerShell-Profile (siehe [noch nicht erstellt]) sind für diese Zwecke hervorragend geeignet.

Aliase online lesen: https://riptutorial.com/de/powershell/topic/5287/aliase

Kapitel 4: Amazon Web Services (AWS) - einfacher Speicherdienst (S3)

Einführung

Dieser Dokumentationsabschnitt konzentriert sich auf die Entwicklung gegen den Simple Storage Service (S3) von Amazon Web Services (AWS). S3 ist wirklich ein einfacher Dienst, mit dem Sie interagieren können. Sie erstellen S3 "Buckets", die null oder mehr "Objekte" enthalten können. Nachdem Sie einen Bucket erstellt haben, können Sie Dateien oder beliebige Daten als "Objekt" in den S3-Bucket hochladen. Sie referenzieren S3-Objekte innerhalb eines Buckets anhand des "Schlüssels" (Name) des Objekts.

Parameter

Parameter	Einzelheiten
BucketName	Der Name des AWS S3-Buckets, mit dem Sie arbeiten.
CannedACLName	Der Name der integrierten (vordefinierten) Zugriffssteuerungsliste (ACL), die dem S3-Bucket zugeordnet wird.
Datei	Der Name einer Datei im lokalen Dateisystem, die in einen AWS S3-Bucket hochgeladen wird.

Examples

Erstellen Sie einen neuen S3-Bucket

```
New-S3Bucket -BucketName trevor
```

Der Bucket-Name des Simple Storage Service (S3) muss global eindeutig sein. Das bedeutet, dass, wenn jemand anderes den Bucket-Namen verwendet hat, den Sie verwenden möchten, Sie einen neuen Namen auswählen müssen.

Laden Sie eine lokale Datei in einen S3-Bucket hoch

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'
Write-S3Object -BucketName powershell -File myfile.txt
```

Das Hochladen von Dateien aus Ihrem lokalen Dateisystem in AWS S3 ist mit dem Befehl `Write-S3Object` einfach. In der einfachsten Form müssen Sie lediglich den Parameter `-BucketName` angeben, um anzugeben, in welchen S3-Bucket Sie eine Datei hochladen möchten, und den

Parameter `-File` , der den relativen oder absoluten Pfad zur gewünschten lokalen Datei angibt in den S3-Bucket hochladen.

Löschen Sie einen S3-Bucket

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force
Remove-S3Bucket -BucketName powershell -Force
```

Um einen S3-Bucket zu entfernen, müssen Sie zuerst alle im Bucket gespeicherten S3-Objekte entfernen, sofern Sie dazu die Berechtigung haben. Im obigen Beispiel `Remove-S3Object` wir eine Liste aller Objekte in einem Bucket ab und `Remove-S3Object` sie dann in den Befehl `Remove-S3Object` , um sie zu löschen. Nachdem alle Objekte entfernt wurden, können `Remove-S3Bucket` Befehl " `Remove-S3Bucket` , um den Bucket zu löschen.

Amazon Web Services (AWS) - einfacher Speicherdienst (S3) online lesen: https://riptutorial.com/de/powershell/topic/9579/amazon-web-services--aws----einfacher-speicherdienst--s3-

Kapitel 5: Anonymisieren Sie IP (v4 und v6) in einer Textdatei mit Powershell

Einführung

Regex für IPv4 und IPv6 bearbeiten und durch gefälschte IP-Adresse in einer gelesenen Protokolldatei ersetzen

Examples

IP-Adresse in Textdatei anonymisieren

```
# Read a text file and replace the IPv4 and IPv6 by fake IP Address


# Describe all variables
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "((([0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})|(([0-9A-Fa-
f]{1,4}:){4}:([0-9A-Fa-f]{1,4}:){0,2}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-
f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-
5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(([0-9A-Fa-f]{1,4}:){0,5}:((b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(::([0-9A-Fa-
f]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}:){0,5}[0-9A-Fa-f]{1,4})|(::([0-9A-Fa-
f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){1,7}:))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# Replace matching IPv4 from sourcefile and creating a temp file IPV4.txt
Get-Content $SourceFile | Foreach-Object {$_ -replace $Regex_v4, $Anonym_v4} | Set-Content
$IPv4File

# Replace matching IPv6 from IPV4.txt and creating a temp file ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$_ -replace $Regex_v6, $Anonym_v6} | Set-Content
$DestFile

# Delete temp IPV4.txt file
Remove-Item $IPv4File

# Rename ANONYM.txt in sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
    }
```

```
    Else
    {
    Rename-Item $DestFile -NewName $AnonymFile
}
```

Anonymisieren Sie IP (v4 und v6) in einer Textdatei mit Powershell online lesen: https://riptutorial.com/de/powershell/topic/9171/anonymisieren-sie-ip--v4-und-v6--in-einer-textdatei-mit-powershell

Kapitel 6: Anweisung wechseln

Einführung

Mit einer switch-Anweisung kann eine Variable auf Gleichheit mit einer Liste von Werten getestet werden. Jeder Wert ist ein *Fall* genannt, und die Variable *umgeschaltet* wird auf für jedes Schaltergehäuse überprüft. Damit können Sie ein Skript schreiben, das aus einer Reihe von Optionen auswählen kann, ohne dass Sie dazu eine lange Reihe von if-Anweisungen schreiben müssen.

Bemerkungen

In diesem Thema wird die ***switch-Anweisung beschrieben,*** die zum Verzweigen des Skriptflusses verwendet wird. Verwechseln Sie es nicht mit ***Schalterparametern,*** die in Funktionen als boolesche Flags verwendet werden.

Examples

Einfacher Schalter

Switch-Anweisungen vergleichen einen einzelnen Testwert mit mehreren Bedingungen und führen alle zugehörigen Aktionen für erfolgreiche Vergleiche aus. Dies kann zu mehreren Übereinstimmungen / Aktionen führen.

Bei folgendem Wechsel ...

```
switch($myValue)
{
    'First Condition'    { 'First Action' }
    'Second Condition'   { 'Second Action' }
}
```

`'First Action'` wird ausgegeben, wenn `$myValue` als `'First Condition'` .

`'Section Action'` wird ausgegeben, wenn `$myValue` als `'Second Condition'` .

Es wird nichts ausgegeben, wenn `$myValue` keine der Bedingungen erfüllt.

Wechselanweisung mit Regex-Parameter

Mit `-Regex` Parameter `-Regex` können switch-Anweisungen einen Abgleich mit regulären Ausdrücken mit Bedingungen durchführen.

Beispiel:

```
switch -Regex ('Condition')
```

```
{
  'Con\D+ion'    {'One or more non-digits'}
  'Conditio*$'   {'Zero or more "o"'}
  'C.ndition'    {'Any single char.'}
  '^C\w+ition$'  {'Anchors and one or more word chars.'}
  'Test'         {'No match'}
}
```

Ausgabe:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Einfacher Schalter mit Pause

Das Schlüsselwort `break` kann in switch-Anweisungen zum Beenden der Anweisung verwendet werden, bevor alle Bedingungen ausgewertet werden.

Beispiel:

```
switch('Condition')
{
  'Condition'
  {
    'First Action'
  }
  'Condition'
  {
    'Second Action'
    break
  }
  'Condition'
  {
    'Third Action'
  }
}
```

Ausgabe:

```
First Action
Second Action
```

Aufgrund des Schlüsselworts `break` in der zweiten Aktion wird die dritte Bedingung nicht ausgewertet.

Wechselanweisung mit Platzhalterparameter

Mit `-Wildcard` Parameter `-Wildcard` können switch-Anweisungen einen Platzhalterabgleich mit Bedingungen durchführen.

Beispiel:

```
switch -Wildcard ('Condition')
{
    'Condition'           {'Normal match'}
    'Condit*'             {'Zero or more wildcard chars.'}
    'C[aoc]ndit[f-l]on'   {'Range and set of chars.'}
    'C?ndition'           {'Single char. wildcard'}
    'Test*'               {'No match'}
}
```

Ausgabe:

```
Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard
```

Wechselanweisung mit genauem Parameter

Der Parameter `-Exact` erzwingt, dass switch-Anweisungen einen exakten, von der Groß- und Kleinschreibung abhängigen Abgleich mit Zeichenfolgenbedingungen durchführen.

Beispiel:

```
switch -Exact ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
  '^*ondition$' {'Fourth Action'}
  'Conditio*'   {'Fifth Action'}
}
```

Ausgabe:

```
First Action
Second Action
Third Action
```

Die erste bis dritte Aktion wird ausgeführt, weil ihre zugehörigen Bedingungen mit der Eingabe übereinstimmen. Die Regex- und Platzhalterzeichenfolgen in der vierten und fünften Bedingung stimmen nicht überein.

Beachten Sie, dass die vierte Bedingung auch mit der Eingabezeichenfolge übereinstimmen würde, wenn der Abgleich mit regulären Ausdrücken durchgeführt wurde, in diesem Fall jedoch ignoriert wurde, da dies nicht der Fall ist.

Switch-Anweisung mit CaseSensitive-Parameter

Mit `-CaseSensitive` Parameter `-CaseSensitive` werden Switch-Anweisungen `-CaseSensitive` , um eine genaue, von der Groß- und Kleinschreibung abhängige Übereinstimmung mit Bedingungen durchzuführen.

Beispiel:

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
}
```

Ausgabe:

```
Second Action
```

Die zweite Aktion ist die einzige Aktion, die ausgeführt wird, da sie die einzige Bedingung ist, die genau mit der Zeichenfolge `'Condition'` übereinstimmt `'Condition'` wenn die Groß- / Kleinschreibung berücksichtigt wird.

Wechselanweisung mit Dateiparameter

Mit `-file` Parameter `-file` kann die switch-Anweisung Eingaben von einer Datei empfangen. Jede Zeile der Datei wird von der switch-Anweisung ausgewertet.

Beispieldatei `input.txt` :

```
condition
test
```

Beispiel für eine Wechselanweisung:

```
switch -file input.txt
{
  'condition' {'First Action'}
  'test'      {'Second Action'}
  'fail'      {'Third Action'}
}
```

Ausgabe:

```
First Action
Second Action
```

Einfacher Switch mit Standardbedingung

Mit dem `Default` wird eine Aktion ausgeführt, wenn keine anderen Bedingungen mit dem Eingabewert übereinstimmen.

Beispiel:

```
switch('Condition')
{
```

```
  'Skip Condition'
  {
    'First Action'
  }
  'Skip This Condition Too'
  {
    'Second Action'
  }
  Default
  {
    'Default Action'
  }
}
```

Ausgabe:

```
Default Action
```

Anweisung mit Ausdrücken wechseln

Bedingungen können auch Ausdrücke sein:

```
$myInput = 0

switch($myInput) {
    # because the result of the expression, 4,
    # does not equal our input this block should not be run.
    (2+2)  { 'True. 2 +2 = 4' }

    # because the result of the expression, 0,
    # does equal our input this block should be run.
    (2-2) { 'True. 2-2 = 0' }

    # because our input is greater than -1 and is less than 1
    # the expression evaluates to true and the block should be run.
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}

#Output
True. 2-2 = 0
True. Value is 0
```

Anweisung wechseln online lesen: https://riptutorial.com/de/powershell/topic/1174/anweisung-wechseln

Kapitel 7: Arbeiten mit der PowerShell-Pipeline

Einführung

In PowerShell wird ein Objekt-Pipelining-Modell eingeführt, mit dem Sie ganze Objekte durch die Pipeline an verbrauchende Commandlets oder (mindestens) die Ausgabe senden können. Im Gegensatz zum klassischen String-basierten Pipelining müssen Informationen in Pipe-Objekten nicht an bestimmten Positionen sein. Commandlets können erklären, mit Objekten aus der Pipeline als Eingabe zu interagieren, während Rückgabewerte automatisch an die Pipeline gesendet werden.

Syntax

- BEGIN Der erste Block. Einmal am Anfang ausgeführt. Die Pipeline-Eingabe hier ist $ null, da sie nicht festgelegt wurde.
- PROCESS Der zweite Block. Wird für jedes Element der Pipeline ausgeführt. Der Pipeline-Parameter entspricht dem aktuell verarbeiteten Element.
- ENDE Letzter Block. Einmal ausgeführt am Ende. Der Pipeline-Parameter ist gleich dem letzten Element der Eingabe, da er seit der Einstellung nicht geändert wurde.

Bemerkungen

In den meisten Fällen handelt es sich bei der Eingabe der Pipeline um ein Array von Objekten. Obwohl das Verhalten des Blocks `PROCESS{}` dem `foreach{}` Block ähnelt, erfordert das Überspringen eines Elements im Array einen anderen Prozess.

Wenn Sie, wie in `foreach{}` , innerhalb des `PROCESS{}` Blocks `continue` , würde dies die Pipeline zerstören und alle folgenden Anweisungen einschließlich des `END{}` Blocks überspringen. Verwenden Sie stattdessen `return` , um den Block `PROCESS{}` für das aktuelle Element zu beenden und zum nächsten zu wechseln.

In einigen Fällen müssen die Ergebnisse von Funktionen mit unterschiedlicher Codierung ausgegeben werden. Die Codierung der Ausgabe der CmdLets wird durch die Variable `$OutputEncoding` gesteuert. Wenn die Ausgabe für native Anwendungen in einer Pipeline abgelegt werden soll, `$OutputEncoding = [Console]::OutputEncoding` es sich, die Kodierung entsprechend dem Ziel `$OutputEncoding = [Console]::OutputEncoding`

Zusätzliche referenzen:

Blogartikel mit mehr `$OutputEncoding`
https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/

Examples

Schreibfunktionen mit Advanced Lifecycle

Dieses Beispiel zeigt, wie eine Funktion Eingaben mit Pipelines akzeptieren und effizient wiederholen kann.

Beachten Sie , dass die `begin` und `end` Strukturen der Funktion sind optional , wenn Pipelining, aber das `process` ist erforderlich, wenn mit `ValueFromPipeline` oder `ValueFromPipelineByPropertyName` .

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

Ausgabe:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Grundlegende Pipeline-Unterstützung in Funktionen

Dies ist ein Beispiel für eine Funktion mit einer möglichst einfachen Unterstützung für das Pipelining.
Für jede Funktion mit Pipeline-Unterstützung muss mindestens ein Parameter festgelegt sein, für den ParameterAttribute `ValueFromPipeline` oder `ValueFromPipelineByPropertyName` festgelegt ist (siehe unten).

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)]  # This sets the ParameterAttribute
        [String]$Input
```

```
    )
    Write-Host $Input
}

$foo = 'Hello World!'

$foo | Write-FromPipeline
```

Ausgabe:

```
Hello World!
```

Hinweis: In PowerShell 3.0 und höher werden Standardwerte für ParameterAttributes unterstützt. In früheren Versionen müssen Sie `ValueFromPipeline=$true` angeben.

Arbeitskonzept der Pipeline

In einer Pipeline-Reihe verläuft jede Funktion parallel zu den anderen, wie parallele Threads. Das erste bearbeitete Objekt wird zur nächsten Pipeline übertragen und die nächste Bearbeitung wird sofort in einem anderen Thread ausgeführt. Dies erklärt den hohen Geschwindigkeitsgewinn im Vergleich zum Standard `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. Schritt - kopiere die erste Datei (in `Copy-file` Thread)
2. Schritt - Zweite Datei kopieren (in `Copy-file` Thread) und gleichzeitig Erste `Encrypt-File` (in `Encrypt-File`)
3. Schritt - dritte Datei kopieren (in `Copy-file` Thread) und zweite Datei gleichzeitig verschlüsseln (in `Encrypt-File`) und gleichzeitig `get-Md5` der ersten (in `Get-Md5`)

Arbeiten mit der PowerShell-Pipeline online lesen:
https://riptutorial.com/de/powershell/topic/3937/arbeiten-mit-der-powershell-pipeline

Kapitel 8: Archivmodul

Einführung

Das Archivmodul `Microsoft.PowerShell.Archive` bietet Funktionen zum Speichern von Dateien in ZIP-Archiven (`Compress-Archive`) und Extrahieren (`Expand-Archive`). Dieses Modul ist in PowerShell 5.0 und höher verfügbar.

In früheren Versionen von PowerShell können die Community Extensions oder .NET System.IO.Compression.FileSystem verwendet werden.

Syntax

- **Expand-Archiv** / **Komprimieren-Archiv**
- **-Pfad**
  - der Pfad der zu komprimierenden Datei (en) (Compress-Archive) oder der Pfad des Archivs zum Entpacken der Datei (en) (Expand-Archive)
  - Es gibt mehrere andere Pfadoptionen, siehe unten.
- **-DestinationPath** (optional)
  - Wenn Sie diesen Pfad nicht angeben, wird das Archiv im aktuellen Arbeitsverzeichnis erstellt (Compress-Archive) oder der Inhalt des Archivs wird in das aktuelle Arbeitsverzeichnis entpackt (Expand-Archiv).

Parameter

Parameter	Einzelheiten
CompressionLevel	*(Nur Compress-Archive)* `NoCompression` Sie die Komprimierungsstufe auf `Fastest` , `Optimal` oder `NoCompression`
Bestätigen	Fordert vor dem Ausführen zur Bestätigung auf
Macht	Erzwingt die Ausführung des Befehls ohne Bestätigung
LiteralPath	Pfad, der buchstäblich verwendet wird, *keine Platzhalterzeichen* werden verwendet ， um mehrere Pfade anzugeben
Pfad	Pfad, der Platzhalterzeichen enthalten kann, verwenden Sie ， um mehrere Pfade anzugeben
Aktualisieren	*(Nur Compress-Archive)* Bestehendes Archiv aktualisieren
Was, wenn	Simulieren Sie den Befehl

Bemerkungen

Weitere Informationen finden Sie unter MSDN Microsoft.PowerShell.Archive (5.1)

Examples

Compress-Archiv mit Platzhalter

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath
C:\Archives\Documents.zip
```

Dieser Befehl:

- Komprimiert alle Dateien in `C:\Documents`
- Verwendet `Optimal` Kompression
- Speichern Sie das resultierende Archiv in `C:\Archives\Documents.zip`
    - `-DestinationPath` fügt `.zip` wenn nicht vorhanden.
    - `-LiteralPath` kann verwendet werden, wenn Sie es ohne `.zip` benennen `-LiteralPath` .

Bestehende ZIP mit Compress-Archiv aktualisieren

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- Dadurch werden alle Dateien `Documents.zip` hinzugefügt oder durch die neuen Dateien aus `C:\Documents`

Extrahieren Sie eine Zip mit Expand-Archive

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- Dadurch werden alle Dateien aus `Documents.zip` in den Ordner `C:\Documents` extrahiert

Archivmodul online lesen: https://riptutorial.com/de/powershell/topic/9896/archivmodul

Kapitel 9: Ausführbare Dateien ausführen

Examples

Konsolenanwendungen

```
PS> console_app.exe
PS> & console_app.exe
PS> Start-Process console_app.exe
```

GUI-Anwendungen

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

GUI-Anwendungen werden in einem anderen Prozess gestartet und geben die Kontrolle sofort an den PowerShell-Host zurück. Manchmal muss die Anwendung die Verarbeitung abschließen, bevor die nächste PowerShell-Anweisung ausgeführt werden muss. Dies kann durch Weiterleiten der Anwendungsausgabe an $ null (3) oder durch Verwendung von Start-Process mit dem Schalter -Wait (5) erreicht werden.

Konsolenströme

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

Stream 2 enthält System.Management.Automation.ErrorRecord-Objekte. Beachten Sie, dass einige Anwendungen wie "git.exe" den "Fehlerstrom" zu Informationszwecken verwenden, die nicht unbedingt Fehler sind. In diesem Fall sollten Sie den Exit-Code am besten prüfen, ob der Fehlerstrom als Fehler interpretiert werden soll.

PowerShell versteht diese Streams: Ausgabe, Fehler, Warnung, Verbose, Debug, Fortschritt. Native Anwendungen verwenden im Allgemeinen nur diese Streams: Ausgabe, Fehler, Warnung.

In PowerShell 5 können alle Streams auf den Standard-Output / Success-Stream umgeleitet werden (6).

In früheren PowerShell-Versionen können nur bestimmte Streams auf den Standard-Output / Success-Stream umgeleitet werden (7). In diesem Beispiel wird der "Fehlerstrom" in den

Ausgabestrom umgeleitet.

Beendigungscodes

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

Hierbei handelt es sich um integrierte PowerShell-Variablen, die zusätzliche Informationen zum neuesten Fehler enthalten. `$LastExitCode` ist der endgültige Exit-Code der zuletzt ausgeführten `$LastExitCode` Anwendung. `$?` und `$Error[0]` ist der letzte von PowerShell erzeugte Fehlerdatensatz.

Ausführbare Dateien ausführen online lesen:
https://riptutorial.com/de/powershell/topic/7707/ausfuhrbare-dateien-ausfuhren

Kapitel 10: Automatische Variablen

Einführung

Automatische Variablen werden von Windows PowerShell erstellt und verwaltet. Man hat die Möglichkeit, eine Variable fast jeden Namen im Buch aufzurufen. Die einzigen Ausnahmen sind die Variablen, die bereits von PowerShell verwaltet werden. Diese Variablen sind ohne Zweifel die am häufigsten wiederkehrenden Objekte, die Sie in PowerShell neben Funktionen verwenden (z. B. **$?** - zeigt den Status des letzten Vorgangs an.

Syntax

- `$$` - Enthält das letzte Token in der letzten von der Sitzung empfangenen Zeile.
- `$^` - Enthält das erste Token in der letzten Zeile, die von der Sitzung empfangen wurde.
- `$?` - Enthält den Ausführungsstatus der letzten Operation.
- `$_` - Enthält das aktuelle Objekt in der Pipeline

Examples

$ pid

Enthält die Prozess-ID des aktuellen Hosting-Prozesses.

```
PS C:\> $pid
26080
```

Boolesche Werte

`$true` und `$false` sind zwei Variablen, die logisch TRUE und FALSE darstellen.

Beachten Sie, dass Sie das Dollarzeichen als erstes Zeichen angeben müssen (was sich von C # unterscheidet).

```
$boolExpr = "abc".Length -eq 3 # length of "abc" is 3, hence $boolExpr will be True
if($boolExpr -eq $true){
    "Length is 3"
}
# result will be "Length is 3"
$boolExpr -ne $true
#result will be False
```

Beachten Sie, dass Sie bei der Verwendung von boolean true / false in Ihrem Code `$true` oder `$false` schreiben. Wenn Powershell jedoch einen boolean zurückgibt, sieht es aus wie `True` oder `False`

$ null

`$null` wird verwendet, um einen abwesenden oder undefinierten Wert darzustellen.
`$null` kann als leerer Platzhalter für leere Werte in Arrays verwendet werden:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Wenn wir dasselbe Array als Quelle für `ForEach-Object` , werden alle drei Elemente (einschließlich $ null) verarbeitet:

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Achtung! Das bedeutet , dass `ForEach-Object` - Prozess **wird** auch `$null` ganz von selbst:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Das ist ein sehr unerwartetes Ergebnis, wenn Sie es mit der klassischen `foreach` Schleife vergleichen:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

$ OFS

Variable mit dem Namen Ausgabefeldtrennzeichen enthält einen Zeichenfolgewert, der beim Konvertieren eines Arrays in einen String verwendet wird. Standardmäßig ist `$OFS = " "` (*ein Leerzeichen*), es kann jedoch geändert werden:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # default OFS will be used
1 2 3
PS C:\> $OFS = ",." # we change OFS to comma and dot
PS C:\> "$array"
1,.2,.3
```

$ _ / $ PSItem

Enthält das Objekt / Element, das gerade von der Pipeline verarbeitet wird.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` und `$_` sind identisch und können austauschbar verwendet werden, aber `$_` wird bei weitem am häufigsten verwendet.

$?

Enthält den Status der letzten Operation. Wenn es keinen Fehler gibt, wird es auf `True` :

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

Wenn ein Fehler vorliegt, wird er auf `False` :

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script
file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ wrt-host
+ ~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

$ Fehler

Array der neuesten Fehlerobjekte. Der erste im Array ist der neueste:

```
PS C:\> throw "Error" # resulting output will be in red font
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error

PS C:\> $error[0] # resulting output will be normal string (not red    )
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error
```

Hinweise zur Verwendung: Wenn Sie die Variable `$error` in einem Format-Cmdlet (z. B. format-list) verwenden, müssen Sie die `-Force` . Andernfalls gibt das Format-Cmdlet den `$error` Inhalt wie oben angegeben aus.

`$Error.Remove($Error[0])` können über zB `$Error.Remove($Error[0])` .

Automatische Variablen online lesen:
https://riptutorial.com/de/powershell/topic/5353/automatische-variablen

Kapitel 11: Automatische Variablen - Teil 2

Einführung

Im Thema "Automatische Variablen" sind bereits 7 Beispiele aufgeführt, und wir können keine weiteren hinzufügen. Dieses Thema enthält eine Fortsetzung der automatischen Variablen.

Automatische Variablen sind Variablen, die Statusinformationen für PowerShell speichern. Diese Variablen werden von Windows PowerShell erstellt und verwaltet.

Bemerkungen

Nicht sicher, ob dies der beste Weg ist, automatische Variablen zu dokumentieren, aber das ist besser als nichts. Bitte kommentieren Sie, wenn Sie einen besseren Weg finden :)

Examples

$ PSVersionTable

Enthält eine schreibgeschützte Hashtabelle (Constant, AllScope), in der Details zu der Version von PowerShell angezeigt werden, die in der aktuellen Sitzung ausgeführt wird.

```
$PSVersionTable    #this call results in this:
Name                           Value
----                           -----
PSVersion                      5.0.10586.117
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.10586.117
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

Der schnellste Weg, eine Version von PowerShell zum Laufen zu bringen:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----  -----  -----  --------
5      0      10586  117
```

Automatische Variablen - Teil 2 online lesen:
https://riptutorial.com/de/powershell/topic/8639/automatische-variablen---teil-2

Kapitel 12: Bedingte Logik

Syntax

- if (Ausdruck) {}
- if (Ausdruck) {} else {}
- if (Ausdruck) {} elseif (Ausdruck) {}
- if (Ausdruck) {} elseif (Ausdruck) {} else {}

Bemerkungen

Siehe auch Vergleichsoperatoren , die in bedingten Ausdrücken verwendet werden können.

Examples

wenn, sonst und sonst wenn

Powershell unterstützt standardmäßige bedingte Logikoperatoren, ähnlich wie viele Programmiersprachen. Dadurch können bestimmte Funktionen oder Befehle unter bestimmten Umständen ausgeführt werden.

Bei einem `if` die Befehle in den Klammern (`{}`) nur ausgeführt, wenn die Bedingungen innerhalb des if (`()`) erfüllt sind

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

Sie können auch etwas `else` . Hier werden die `else` Befehle ausgeführt, wenn die `if` Bedingungen **nicht** erfüllt sind:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

oder ein `elseif` . Ein else if führt die Befehle aus, wenn die `if` Bedingungen nicht erfüllt sind und die `elseif` Bedingungen erfüllt sind:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
```

```
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Beachten Sie die Verwendung von `-eq` (Gleichheit) mit CmdLet und nicht `=` oder `==` wie viele andere Sprachen für die Gleichheit.

Negation

Sie können einen booleschen Wert negieren, dh, eine `if` -Anweisung eingeben, `if` eine Bedingung falsch ist und nicht wahr. Dies kann mit dem `-Not` -Not `-Not` durchgeführt werden

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

Sie können auch verwenden `!` :

```
$test = "test"
if (!($test -eq "test2")){
    Write-Host "if condition not met"
}
```

Es gibt auch den Operator `-ne` (nicht gleich):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Wenn bedingte Abkürzung

Wenn Sie die Abkürzung verwenden möchten, können Sie die bedingte Logik mit der folgenden Abkürzung verwenden. Nur die Zeichenfolge 'false' wird zu true (2.0) ausgewertet.

```
#Done in Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # this does not run because $boolean is false
    Write-Host "Shorthand If conditions can be nice, just make sure they are always boolean."
}

If($string){
    # This does run because the string is non-zero length
    Write-Host "If the variable is not strictly null or Boolean false, it will evaluate to
true as it is an object or string with length greater than 0."
}

If($emptyString){
```

```
    # This does not run because the string is zero-length
    Write-Host "Checking empty strings can be useful as well."
}

If($null){
    # This does not run because the condition is null
    Write-Host "Checking Nulls will not print this statement."
}
```

Bedingte Logik online lesen: https://riptutorial.com/de/powershell/topic/7208/bedingte-logik

Kapitel 13: Berechnete Eigenschaften

Einführung

Berechnete Eigenschaften in Powershell sind benutzerdefinierte abgeleitete (berechnete) Eigenschaften. Dadurch kann der Benutzer eine bestimmte Eigenschaft auf eine Weise formatieren, die er möchte. Die Berechnung (Ausdruck) kann durchaus alles sein.

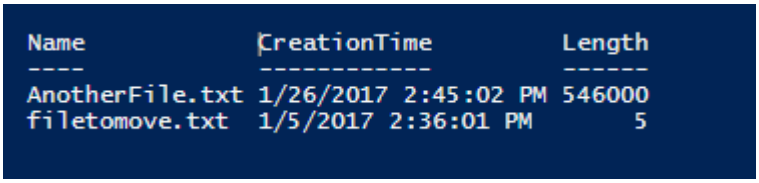
Examples

Dateigröße in KB - Berechnete Eigenschaften anzeigen

Betrachten wir den folgenden Ausschnitt:

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

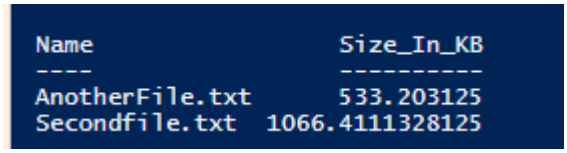
Es wird einfach der Ordnerinhalt mit den ausgewählten Eigenschaften ausgegeben. So etwas wie,



Was ist, wenn ich die Dateigröße in KB anzeigen möchte? Hier sind berechnete Eigenschaften praktisch.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length /
1Kb}}
```

Was produziert,



Der `Expression` enthält die Berechnung für die berechnete Eigenschaft. Und ja, es kann alles sein!

Berechnete Eigenschaften online lesen:
https://riptutorial.com/de/powershell/topic/8913/berechnete-eigenschaften

Kapitel 14: Cmdlet-Benennung

Einführung

CmdLets sollten mit einem Namensschema `<verb>-<noun>` benannt werden, um die Erkennbarkeit zu verbessern.

Examples

Verben

Verben, die zum Benennen von CmdLets verwendet werden, sollten aus den in `Get-Verb` Verben benannt werden

Weitere Informationen zur Verwendung von Verben finden Sie unter Genehmigte Verben für Windows PowerShell

Substantive

Substantive sollten immer einzigartig sein.

Seien Sie konsistent mit den Substantiven. Zum Beispiel benötigt `Find-Package` einen Provider. Das Nomen ist `PackageProvider` nicht `ProviderPackage` .

Cmdlet-Benennung online lesen: https://riptutorial.com/de/powershell/topic/8703/cmdlet-benennung

Kapitel 15: CSV-Analyse

Examples

Grundlegende Verwendung von Import-Csv

Gegeben die folgende CSV-Datei

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

Mit dem Befehl `Import-Csv` können Sie die CSV-Zeilen in PowerShell-Objekte `Import-Csv`

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows

String DateTime            Integer
------ --------            -------
First  2016-12-01T12:00:00 30
Second 2015-11-03T13:00:00 20
Third  2015-12-05T14:00:00 20

> Write-Host $row[0].String1
Third
```

Importieren Sie aus CSV- und Umwandlungseigenschaften in den richtigen Typ

Standardmäßig importiert `Import-CSV` alle Werte als Zeichenfolgen. Um DateTime- und Integer-Objekte zu erhalten, müssen wir sie umwandeln oder analysieren.

`Foreach-Object` :

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #Cast properties
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #Output object
    $_
}
```

Berechnete Eigenschaften verwenden:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
```

```
    @{name="Integer";expression={ [int]$_.Integer }}
```

Ausgabe:

```
String DateTime            Integer
------ --------            -------
First  01.12.2016 12:00:00      30
Second 03.11.2015 13:00:00      20
Third  05.12.2015 14:00:00      20
```

CSV-Analyse online lesen: https://riptutorial.com/de/powershell/topic/5691/csv-analyse

Kapitel 16: Dynamische PowerShell-Parameter

Examples

Dynamischer Parameter "einfach"

In diesem Beispiel wird MyTestFunction ein neuer Parameter hinzugefügt, wenn `$SomeUsefulNumber` größer als 5 ist.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }

}
```

Verwendungszweck:

```
PS >  MyTestFunction -SomeUsefulNumber 3
```

```
SomeUsefulNumber = 3
MandatoryParam1 =

PS >  MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:

PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

Im zweiten Anwendungsbeispiel können Sie deutlich sehen, dass ein Parameter fehlt.

Dynamische Parameter werden auch bei der automatischen Vervollständigung berücksichtigt. Folgendes passiert, wenn Sie am Ende der Zeile Strg + Leerzeichen drücken:

```
PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
ErrorAction          ErrorVariable        OutVariable

PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1      ErrorAction          ErrorVariable        OutVariable
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
```

Dynamische PowerShell-Parameter online lesen:
https://riptutorial.com/de/powershell/topic/6704/dynamische-powershell-parameter

Kapitel 17: Einbetten von verwaltetem Code (C # | VB)

Einführung

In diesem Thema wird kurz beschrieben, wie C # - oder VB .NET-verwalteter Code in einem PowerShell-Skript skriptiert und verwendet werden kann. In diesem Thema werden nicht alle Facetten des Cmdlets Add-Type untersucht.

Weitere Informationen zum Cmdlet Add-Type finden Sie in der MSDN-Dokumentation (für 5.1) hier: https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-Art

Parameter

Parameter	Einzelheiten
-TypeDefinition <String_>	Akzeptiert den Code als Zeichenfolge
-Sprache <String_>	Gibt die Sprache des verwalteten Codes an. Akzeptierte Werte: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

Bemerkungen

Hinzugefügte Typen werden entfernt

In späteren Versionen von PowerShell wurde Remove-TypeData zu den PowerShell-Cmdlet-Bibliotheken hinzugefügt, die das Entfernen eines Typs innerhalb einer Sitzung ermöglichen. Weitere Informationen zu diesem Cmdlet finden Sie hier: https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata

CSharp- und .NET-Syntax

Für die Erfahrung mit .NET ist es selbstverständlich, dass die unterschiedlichen Versionen von C # sich in ihrer Unterstützung für bestimmte Syntax stark unterscheiden können.

Wenn Powershell 1.0 und / oder -Language CSharp verwendet wird, wird für den verwalteten Code .NET 2.0 verwendet, dem eine Reihe von Features fehlen, die C # -Entwickler heutzutage normalerweise ohne weiteres verwenden, wie Generics, Linq und Lambda. Hinzu kommt der formale Polymorphismus, der in späteren Versionen von C # /. NET mit Standardparametern behandelt wird.

Examples

C # Beispiel

In diesem Beispiel wird gezeigt, wie einige grundlegende C # -Dateien in ein PowerShell-Skript eingebettet werden, dem Runspace / der Sitzung hinzugefügt und der Code innerhalb der PowerShell-Syntax verwendet wird.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

VB.NET-Beispiel

In diesem Beispiel wird gezeigt, wie einige grundlegende C # -Dateien in ein PowerShell-Skript eingebettet werden, dem Runspace / der Sitzung hinzugefügt und der Code innerhalb der PowerShell-Syntax verwendet wird.

```
$code = @"
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
            Console.WriteLine("Static Response")
        End Sub
```

```
        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
    End Class
End Namespace
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Einbetten von verwaltetem Code (C # | VB) online lesen:
https://riptutorial.com/de/powershell/topic/9823/einbetten-von-verwaltetem-code--c-sharp---vb-

Kapitel 18: Einführung in Pester

Bemerkungen

Pester ist ein Testframework für PowerShell, mit dem Sie Testfälle für Ihren PowerShell-Code ausführen können. Es kann verwendet werden, um ex auszuführen. Komponententests, mit denen Sie überprüfen können, ob Ihre Module, Skripts usw. wie beabsichtigt funktionieren.

Was ist pester und warum sollte ich mich interessieren?

Examples

Erste Schritte mit Pester

Um mit dem Testen von PowerShell-Code mit dem Pester-Modul beginnen zu können, müssen Sie mit drei Schlüsselwörtern / Befehlen vertraut sein:

- **Beschreiben** : Definiert eine Gruppe von Tests. Alle Pester-Testdateien benötigen mindestens einen Describe-Block.
- **It** : Definiert einen individuellen Test. Sie können mehrere It-Blöcke in einem Descripe-Block enthalten.
- **Sollte** : Der Befehl verify / test. Es wird verwendet, um das Ergebnis zu definieren, das als erfolgreicher Test betrachtet werden soll.

Probe:

```
Import-Module Pester

#Sample function to run tests against
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#Group of tests
Describe "Validate Add-Numbers" {

        #Individual test cases
        It "Should add 2 + 2 to equal 4" {
            Add-Numbers 2 2 | Should Be 4
        }

        It "Should handle strings" {
            Add-Numbers "2" "2" | Should Be 4
        }

        It "Should return an integer"{
            Add-Numbers 2.3 2 | Should BeOfType Int32
        }

}
```

Ausgabe:

```
Describing Validate Add-Numbers
 [+] Should add 2 + 2 to equal 4 33ms
 [+] Should handle strings 19ms
 [+] Should return an integer 23ms
```

Einführung in Pester online lesen: https://riptutorial.com/de/powershell/topic/5753/einfuhrung-in-pester

Kapitel 19: Einführung in Psake

Syntax

- Task - Hauptfunktion zum Ausführen eines Schritts Ihres Buildskripts
- Abhängig - Eigenschaft, die angibt, worauf der aktuelle Schritt angewiesen ist
- default - Es muss immer eine Standardaufgabe vorhanden sein, die ausgeführt wird, wenn keine Anfangsaufgabe angegeben ist
- FormatTaskName - Gibt an, wie jeder Schritt im Ergebnisfenster angezeigt wird.

Bemerkungen

psake ist ein Build-Automatisierungstool, das in PowerShell geschrieben wurde und von Rake (Ruby Make) und Bake (Boo Make) inspiriert ist. Es wird verwendet, um Builds mit Abhängigkeitsmuster zu erstellen. Dokumentation hier verfügbar

Examples

Grundriss

```
Task Rebuild -Depends Clean, Build  {
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Beispiel für FormatTaskName

```
# Will display task as:
# -------- Rebuild --------
# -------- Build --------
FormatTaskName "-------- {0} --------"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
    param($taskName)
    "Running $taskName" - foregroundcolor yellow
}

Task Rebuild -Depends Clean, Build  {
```

```
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Task bedingt ausführen

```
propreties {
    $isOk = $false
}

# By default the Build task won't run, unless there is a param $true
Task Build -precondition { return $isOk } {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

ContinueOnError

```
Task Build -depends Clean {
   "Build"
 }

Task Clean -ContinueOnError {
   "Clean"
    throw "throw on purpose, but the task will continue to run"
 }

Task default -Depends Build
```

Einführung in Psake online lesen: https://riptutorial.com/de/powershell/topic/5019/einfuhrung-in-psake

Kapitel 20: Eingebaute Variablen

Einführung

PowerShell bietet eine Vielzahl nützlicher "automatischer" (eingebauter) Variablen. Bestimmte automatische Variablen werden nur unter besonderen Umständen aufgefüllt, während andere global verfügbar sind.

Examples

$ PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

In diesem Beispiel wird die Liste der untergeordneten Elemente (Verzeichnisse und Dateien) aus dem Ordner abgerufen, in dem sich die Skriptdatei befindet.

Die automatische Variable `$PSScriptRoot` ist `$null` wenn sie außerhalb einer PowerShell-`$PSScriptRoot` verwendet wird. Bei Verwendung *in* einem PowerShell-Skript wurde automatisch der vollständig qualifizierte Dateisystempfad zu dem Verzeichnis definiert, in dem sich die Skriptdatei befindet.

In Windows PowerShell 2.0 ist diese Variable nur in Skriptmodulen (.psm1) gültig. Ab Windows PowerShell 3.0 ist es in allen Skripts gültig.

$ Args

```
$Args
```

Enthält ein Array der nicht deklarierten Parameter und / oder Parameterwerte, die an eine Funktion, ein Skript oder einen Skriptblock übergeben werden. Wenn Sie eine Funktion erstellen, können Sie die Parameter mit dem Schlüsselwort param deklarieren oder eine durch Kommas getrennte Liste von Parametern in Klammern nach dem Funktionsnamen hinzufügen.

In einer Ereignisaktion enthält die Variable $ Args Objekte, die die Ereignisargumente des verarbeiteten Ereignisses darstellen. Diese Variable wird nur im Aktionsblock eines Befehls zur Ereignisregistrierung aufgefüllt. Der Wert dieser Variablen kann auch in der SourceArgs-Eigenschaft des PSEventArgs-Objekts (System.Management.Automation.PSEventArgs) gefunden werden, das Get-Event zurückgibt.

$ PSItem

```
Get-Process | ForEach-Object -Process {
  $PSItem.Name
}
```

Gleich wie `$_` . Enthält das aktuelle Objekt im Pipeline-Objekt. Sie können diese Variable in Befehlen verwenden, die eine Aktion für jedes Objekt oder für ausgewählte Objekte in einer Pipeline ausführen.

$?

```
Get-Process -Name doesnotexist
Write-Host -Object "Was the last operation successful? $?"
```

Enthält den Ausführungsstatus der letzten Operation. Es enthält TRUE, wenn die letzte Operation erfolgreich war, und FALSE, wenn die Operation fehlgeschlagen ist.

$ Fehler

```
Get-Process -Name doesnotexist
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Enthält ein Array von Fehlerobjekten, die die neuesten Fehler darstellen. Der aktuellste Fehler ist das erste Fehlerobjekt im Array ($ Error [0]).

Um zu verhindern, dass ein Fehler zum Array $ Error hinzugefügt wird, verwenden Sie den allgemeinen Parameter ErrorAction mit dem Wert Ignore. Weitere Informationen finden Sie unter "about_CommonParameters" (http://go.microsoft.com/fwlink/?LinkID=113216) .

Eingebaute Variablen online lesen: https://riptutorial.com/de/powershell/topic/8732/eingebaute-variablen

Kapitel 21: Email schicken

Einführung

Eine nützliche Technik für Exchange Server-Administratoren ist das Versenden von E-Mail-Nachrichten über SMTP von PowerShell. Abhängig von der auf Ihrem Computer oder Server installierten Version von PowerShell gibt es mehrere Möglichkeiten, E-Mails über Powershell zu senden. Es gibt eine native Cmdlet-Option, die einfach und benutzerfreundlich ist. Es verwendet das Cmdlet ***Send-MailMessage*** .

Parameter

Parameter	Einzelheiten
Anhänge <String []>	Pfad- und Dateinamen der Dateien, die an die Nachricht angehängt werden sollen. Pfade und Dateinamen können an Send-MailMessage weitergeleitet werden.
Bcc <String []>	E-Mail-Adressen, die eine Kopie einer E-Mail-Nachricht erhalten, jedoch nicht als Empfänger in der Nachricht angezeigt werden. Geben Sie Namen (optional) und die E-Mail-Adresse (erforderlich) ein, z. B. Name someone@example.com oder someone@example.com.
Body <String_>	Inhalt der E-Mail-Nachricht
BodyAsHtml	Es zeigt an, dass der Inhalt im HTML-Format vorliegt.
Cc <String []>	E-Mail-Adressen, die eine Kopie einer E-Mail-Nachricht erhalten. Geben Sie Namen (optional) und die E-Mail-Adresse (erforderlich) ein, z. B. Name someone@example.com oder someone@example.com.
Bescheinigung	Gibt ein Benutzerkonto an, das zum Senden von Nachrichten von der angegebenen E-Mail-Adresse berechtigt ist. Der Standard ist der aktuelle Benutzer. Geben Sie einen Namen wie Benutzer oder Domäne \ Benutzer ein oder geben Sie ein PSCredential-Objekt ein.
DeliveryNotificationOption	Gibt die Benachrichtigungsoptionen für die Zustellung der E-Mail-Nachricht an. Es können mehrere Werte angegeben werden. Übermittlungsbenachrichtigungen werden in einer Nachricht an die im Parameter To angegebene Adresse gesendet. Zulässige Werte: Keine, OnSuccess, OnFailure, Verzögerung, Nie.

Parameter	Einzelheiten
Codierung	Kodierung für Körper und Thema. Zulässige Werte: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Standard, OEM.
Von	E-Mail-Adressen, von denen die E-Mail gesendet wird Geben Sie Namen (optional) und die E-Mail-Adresse (erforderlich) ein, z. B. Name someone@example.com oder someone@example.com.
Hafen	Alternativer Port auf dem SMTP-Server. Der Standardwert ist 25. Verfügbar unter Windows PowerShell 3.0.
Priorität	Priorität der E-Mail-Nachricht Zulässige Werte: Normal, Hoch, Niedrig.
SmtpServer	Name des SMTP-Servers, der die E-Mail-Nachricht sendet. Der Standardwert ist der Wert der Variablen $ PSEmailServer.
Gegenstand	Betreff der E-Mail-Nachricht
Zu	E-Mail-Adressen, an die die E-Mail gesendet wird Geben Sie die Namen (optional) und die E-Mail-Adresse (erforderlich) ein, z
UseSsl	Verwendet das SSL-Protokoll (Secure Sockets Layer), um eine Verbindung zum Remote-Computer herzustellen, um E-Mails zu senden

Examples

Einfache Send-MailMessage

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -
SmtpServer smtp.com
```

Send-MailMessage mit vordefinierten Parametern

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments =  @('C:\files\samplefile1.txt','C:\files\samplefile2.txt')
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
```

```
    Priority = 'High'
    SmtpServer = 'smtp.com'
    UseSSL = $True
}

# Notice: Splatting requires @ instead of $ in front of variable name
Send-MailMessage @parameters
```

SMTPClient - E-Mail mit Textdatei im Nachrichtentext

```
# Define the txt which will be in the email body
$Txt_File = "c:\file.txt"

function Send_mail {
    #Define Email settings
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)

}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

Email schicken online lesen: https://riptutorial.com/de/powershell/topic/2040/email-schicken

Kapitel 22: Erkennung von Amazon Web Services (AWS)

Einführung

Amazon Rekognition ist ein Dienst, mit dem Sie die Bildanalyse einfach in Ihre Anwendungen einfügen können. Mit Rekognition können Sie Objekte, Szenen und Gesichter in Bildern erkennen. Sie können Gesichter auch suchen und vergleichen. Mit der Rekognition-API können Sie Ihren Anwendungen schnell erweiterte, lernbasierte visuelle Suche und Bildklassifizierung hinzufügen.

Examples

Erkennen von Bildbeschriftungen mit AWS Recognition

```
$BucketName = 'trevorrekognition'
$FileName = 'kitchen.jpg'

New-S3Bucket -BucketName $BucketName
Write-S3Object -BucketName $BucketName -File $FileName
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName

$REKResult.Labels
```

Nachdem Sie das obige Skript ausgeführt haben, sollten in Ihrem PowerShell-Host Ergebnisse gedruckt werden, die ungefähr wie folgt aussehen:

```
RESULTS:

Confidence Name
---------- ----
86.87605   Indoors
86.87605   Interior Design
86.87605   Room
77.4853    Kitchen
77.25354   Housing
77.25354   Loft
66.77325   Appliance
66.77325   Oven
```

Wenn Sie das AWS PowerShell-Modul in Verbindung mit dem AWS Rekognition-Dienst verwenden, können Sie Beschriftungen in einem Bild erkennen, z. B. das Identifizieren von Objekten in einem Raum, Attribute der aufgenommenen Fotos und den entsprechenden Vertrauensgrad, den AWS Rekognition für jedes dieser Attribute aufweist.

Mit dem Befehl `Find-REKLabel` können Sie eine Suche nach diesen Attributen / Bezeichnungen `Find-REKLabel` . Während Sie während des API-Aufrufs Bildinhalte als Byte-Array bereitstellen können, ist es eine bessere Methode, Ihre Bilddateien in einen AWS S3-Bucket hochzuladen und dann den Rekognition-Service auf die zu analysierenden S3-Objekte zu verweisen. Das obige

Beispiel zeigt, wie Sie dies erreichen können.

Vergleichen Sie die Gesichtsähnlichkeit mit AWS Rekognition

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg

### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

Das oben bereitgestellte Beispielskript sollte zu folgenden Ergebnissen führen:

```
Face                                Similarity
----                                ----------
Amazon.Rekognition.Model.ComparedFace 90
```

Mit dem AWS Rekognition-Dienst können Sie einen Gesichtsvergleich zwischen zwei Fotos durchführen. Die Nutzung dieses Dienstes ist recht unkompliziert. Laden Sie einfach zwei Bilddateien, die Sie vergleichen möchten, in einen AWS S3-Bucket hoch. `Compare-REKFace` Befehl `Compare-REKFace` , ähnlich dem oben angegebenen Beispiel. Natürlich müssen Sie Ihren eigenen, weltweit eindeutigen S3-Bucket-Namen und -Dateinamen angeben.

Erkennung von Amazon Web Services (AWS) online lesen:
https://riptutorial.com/de/powershell/topic/9581/erkennung-von-amazon-web-services--aws-

Kapitel 23: Erstellen von klassenbasierten DSC-Ressourcen

Einführung

Ab PowerShell Version 5.0 können Sie PowerShell-Klassendefinitionen zum Erstellen von DSC-Ressourcen (Desired State Configuration) verwenden.

Für die `[DscResource()]` DSC-Ressourcen gibt es ein `[DscResource()]` Attribut, das auf die Klassendefinition angewendet wird, und eine `[DscProperty()]` Ressource, um die vom DSC-Ressourcenbenutzer konfigurierbaren Eigenschaften `[DscProperty()]` .

Bemerkungen

Eine klassenbasierte DSC-Ressource muss:

- Mit dem Attribut `[DscResource()]`
- Definieren Sie eine `Test()` Methode, die `[bool]` zurückgibt.
- Definieren Sie eine `Get()` Methode, die ihren eigenen Objekttyp zurückgibt (z. B. `[Ticket]`).
- Definieren Sie eine `Set()` Methode, die `[void]` zurückgibt.
- Mindestens eine `Key` DSC-Eigenschaft

Nach dem Erstellen einer klassenbasierten PowerShell DSC-Ressource muss diese aus einem Modul mithilfe einer Modulmanifestdatei (.psd1) "exportiert" werden. Innerhalb des `DscResourcesToExport` wird der `DscResourcesToExport` verwendet, um ein Array von DSC-Ressourcen (Klassennamen) für das Exportieren aus dem Modul zu deklarieren. Dadurch können Konsumenten des DSC-Moduls die klassenbasierten Ressourcen innerhalb des Moduls "sehen".

Examples

Erstellen Sie eine DSC-Ressource-Skeleton-Klasse

```
[DscResource()]
class File {
}
```

In diesem Beispiel wird veranschaulicht, wie der äußere Abschnitt einer PowerShell-Klasse erstellt wird, die eine DSC-Ressource deklariert. Sie müssen noch den Inhalt der Klassendefinition ausfüllen.

DSC-Ressourcenskelett mit Schlüsseleigenschaft

```
[DscResource()]
class Ticket {
```

```
  [DscProperty(Key)]
  [string] $TicketId
}
```

Eine DSC-Ressource muss mindestens eine Schlüsseleigenschaft deklarieren. Die Schlüsseleigenschaft identifiziert die Ressource eindeutig von anderen Ressourcen. Angenommen, Sie erstellen eine DSC-Ressource, die ein Ticket in einem Ticketsystem darstellt. Jedes Ticket würde eindeutig mit einer Ticket-ID dargestellt.

Jede Eigenschaft, die dem *Benutzer* der DSC-Ressource `[DscProperty()]` muss mit dem Attribut `[DscProperty()]` werden. Dieses Attribut akzeptiert einen `key` , um anzuzeigen, dass die Eigenschaft ein Schlüsselattribut für die DSC-Ressource ist.

DSC-Ressource mit obligatorischer Eigenschaft

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  [DscProperty(Mandatory)]
  [string] $Subject
}
```

Beim Erstellen einer DSC-Ressource werden Sie häufig feststellen, dass nicht jede einzelne Eigenschaft obligatorisch sein sollte. Es gibt jedoch einige grundlegende Eigenschaften, die Sie sicherstellen möchten, dass sie vom Benutzer der DSC-Ressource konfiguriert werden. Sie verwenden den Parameter `Mandatory` des Attributs `[DscResource()]` , um eine Eigenschaft zu deklarieren, die der Benutzer der DSC-Ressource benötigt.

Im obigen Beispiel haben wir einer `Ticket` Ressource, die ein eindeutiges Ticket in einem Ticketsystem darstellt, eine `Subject` Eigenschaft hinzugefügt und diese als `Mandatory` Eigenschaft festgelegt.

DSC-Ressource mit erforderlichen Methoden

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  # The subject line of the ticket
  [DscProperty(Mandatory)]
  [string] $Subject

  # Get / Set if ticket should be open or closed
  [DscProperty(Mandatory)]
  [string] $TicketState

  [void] Set() {
    # Create or update the resource
  }
```

```
  [Ticket] Get() {
    # Return the resource's current state as an object
    $TicketState = [Ticket]::new()
    return $TicketState
  }

  [bool] Test() {
    # Return $true if desired state is met
    # Return $false if desired state is not met
    return $false
  }
}
```

Dies ist eine vollständige DSC-Ressource, die alle wesentlichen Anforderungen zum Erstellen einer gültigen Ressource veranschaulicht. Die Methodenimplementierungen sind nicht vollständig, sondern sollen die Grundstruktur aufzeigen.

Erstellen von klassenbasierten DSC-Ressourcen online lesen:
https://riptutorial.com/de/powershell/topic/8733/erstellen-von-klassenbasierten-dsc-ressourcen

Kapitel 24: Fehlerbehandlung

Einführung

In diesem Thema werden Fehlertypen und Fehlerbehandlung in PowerShell beschrieben.

Examples

Fehlertypen

Ein Fehler ist ein Fehler, man könnte sich fragen, wie es Typen geben könnte. Nun, bei Powershell fällt der Fehler weitgehend in zwei Kriterien,

- Fehler beim Beenden
- Nicht beendender Fehler

Wie der Name schon sagt, wird durch Beendigungsfehler die Ausführung beendet, und bei einem nicht beendenden Fehler wird die Ausführung mit der nächsten Anweisung fortgesetzt.

> Dies ist wahr, vorausgesetzt, der **$ ErrorActionPreference-** Wert ist default (Continue). $ ErrorActionPreference ist eine Prefrence-Variable, die Powershell mitteilt, was im Falle eines "Non-Terminating" -Fehlers zu tun ist.

Fehler beim Beenden

Ein Beendigungsfehler kann wie unten beschrieben mit einem typischen Try-Catch behandelt werden

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

Das obige Snippet wird ausgeführt und der Fehler wird durch den catch-Block aufgefangen.

Nicht beendender Fehler

Ein nicht terminierender Fehler in der anderen Hand wird jedoch standardmäßig nicht im catch-Block erfasst. Der Grund dafür ist ein nicht beendeter Fehler, der nicht als kritischer Fehler angesehen wird.

```
Try
{
```

```
    Stop-Process -Id 123456
}
Catch
{
    Write-Host "Non-Terminating Error: Invalid Process ID"
}
```

Wenn Sie die Zeile oberhalb der Zeile ausführen, wird die Ausgabe nicht von catch block abgerufen, da der Fehler nicht als kritisch angesehen wird und die Ausführung einfach mit dem nächsten Befehl fortgesetzt wird. Der Fehler wird jedoch in der Konsole angezeigt. Um einen nicht beendenden Fehler zu behandeln, müssen Sie einfach die Fehlerpräferenz ändern.

```
Try
{
    Stop-Process -Id 123456 -ErrorAction Stop
}
Catch
{
    "Non-Terminating Error: Invalid Process ID"
}
```

Mit der aktualisierten Voreinstellung "Fehler" wird dieser Fehler nun als Beendigungsfehler betrachtet und im catch-Block abgefangen.

Abbruch- und nicht beendende Fehler aufrufen:

Das Cmdlet **Write-Error** schreibt den Fehler einfach in das aufrufende Hostprogramm. Es stoppt nicht die Hinrichtung. Where als **Wurf** gibt Ihnen einen Abbruchfehler und stoppt die Ausführung

```
Write-host "Going to try a non terminating Error "
Write-Error "Non terminating"
Write-host "Going to try a terminating Error "
throw "Terminating Error "
Write-host "This Line wont be displayed"
```

Fehlerbehandlung online lesen: https://riptutorial.com/de/powershell/topic/8075/fehlerbehandlung

Kapitel 25: Fortschrittsbalken verwenden

Einführung

Ein Fortschrittsbalken kann verwendet werden, um anzuzeigen, dass sich etwas in einem Prozess befindet. Es ist eine zeitsparende und glatte Funktion, die man haben sollte. Fortschrittsbalken sind beim Debuggen unglaublich nützlich, um herauszufinden, welcher Teil des Skripts ausgeführt wird, und sie sind zufriedenstellend für die Personen, die Skripts ausführen, um zu verfolgen, was passiert. Es ist üblich, einen Fortschritt anzuzeigen, wenn ein Skript lange dauert. Wenn ein Benutzer das Skript startet und nichts passiert, fragt man sich, ob das Skript ordnungsgemäß gestartet wurde.

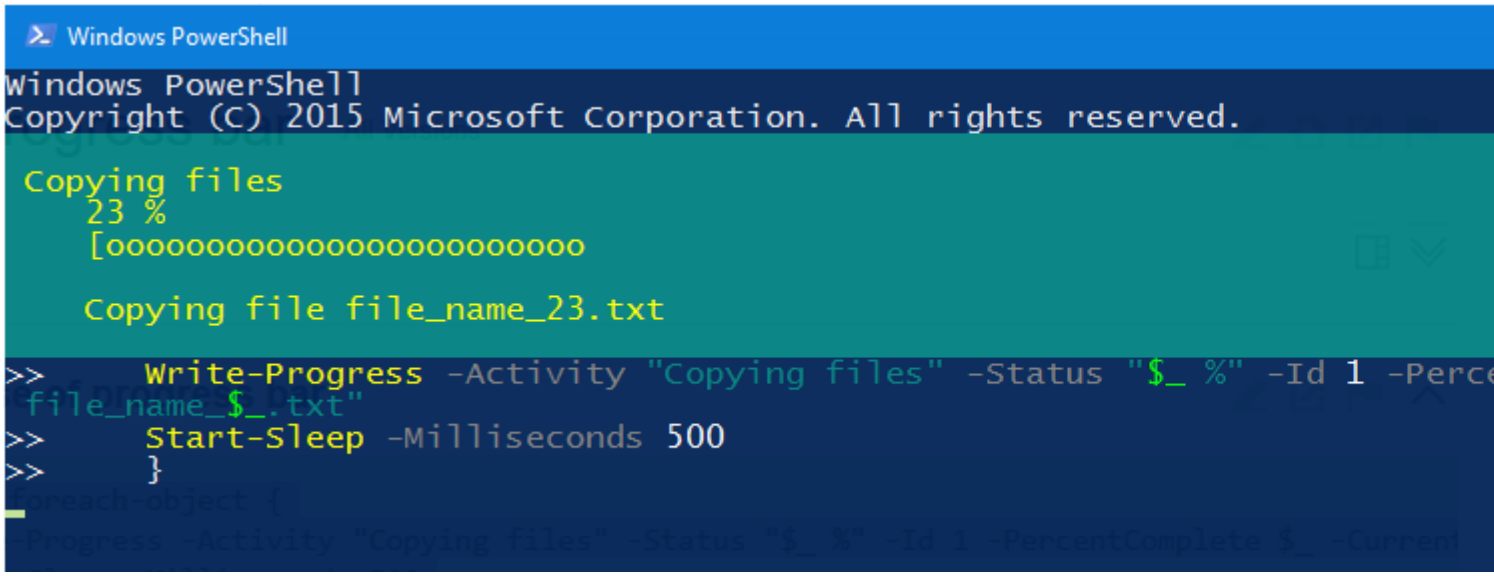
Examples

Einfache Verwendung der Fortschrittsleiste

```
1..100 | ForEach-Object {
        Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -
CurrentOperation "Copying file file_name_$_.txt"
        Start-Sleep -Milliseconds 500    # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
    }
```

Bitte beachten Sie, dass dieses Beispiel der Kürze halber keinen Code enthält (mit `Start-Sleep` simuliert). Es ist jedoch möglich, es direkt als solches auszuführen und dann zu modifizieren und damit zu spielen.

So sieht das Ergebnis in der PS-Konsole aus:



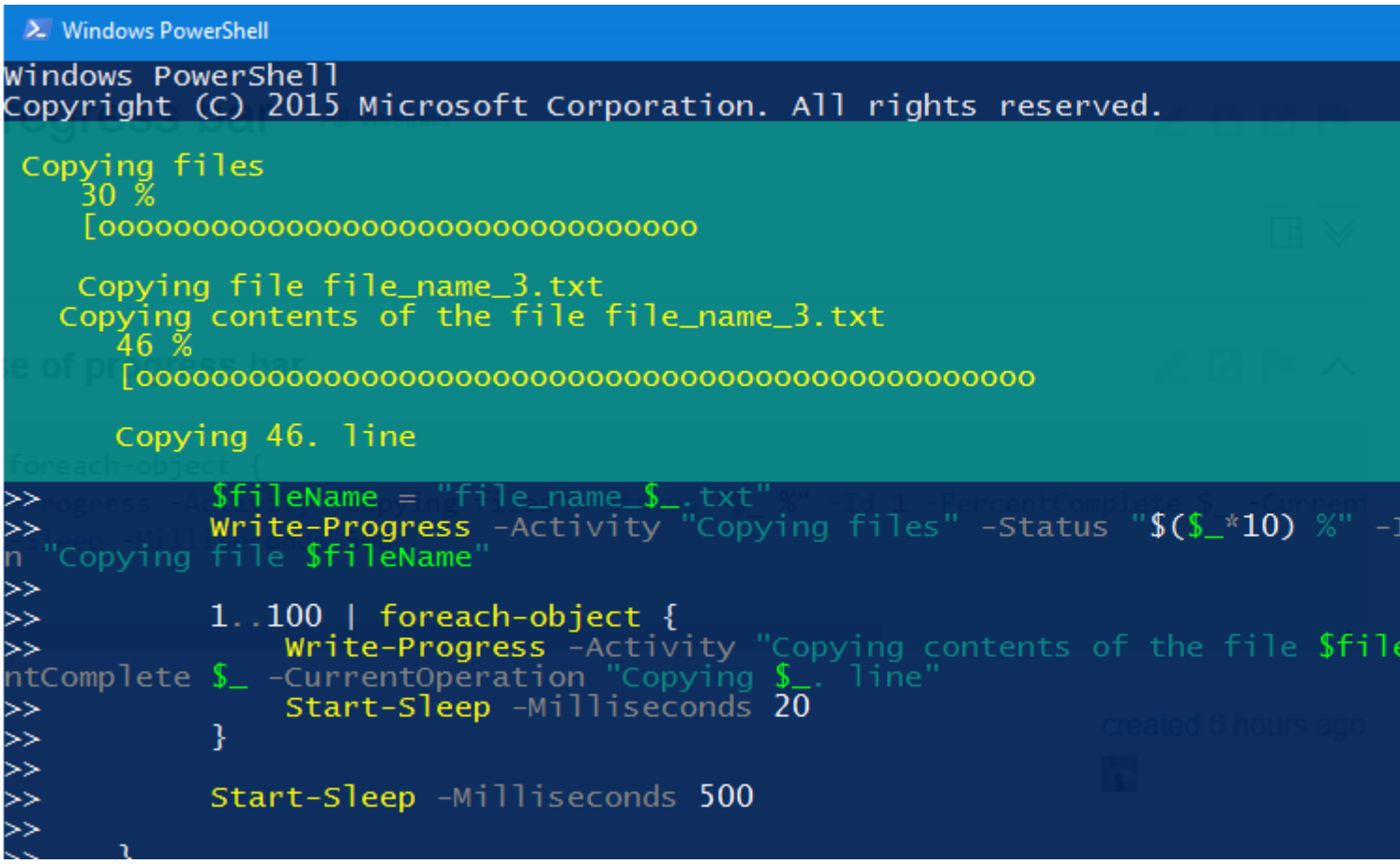
So sieht das Ergebnis in PS ISE aus:

Verwendung des inneren Fortschrittsbalkens

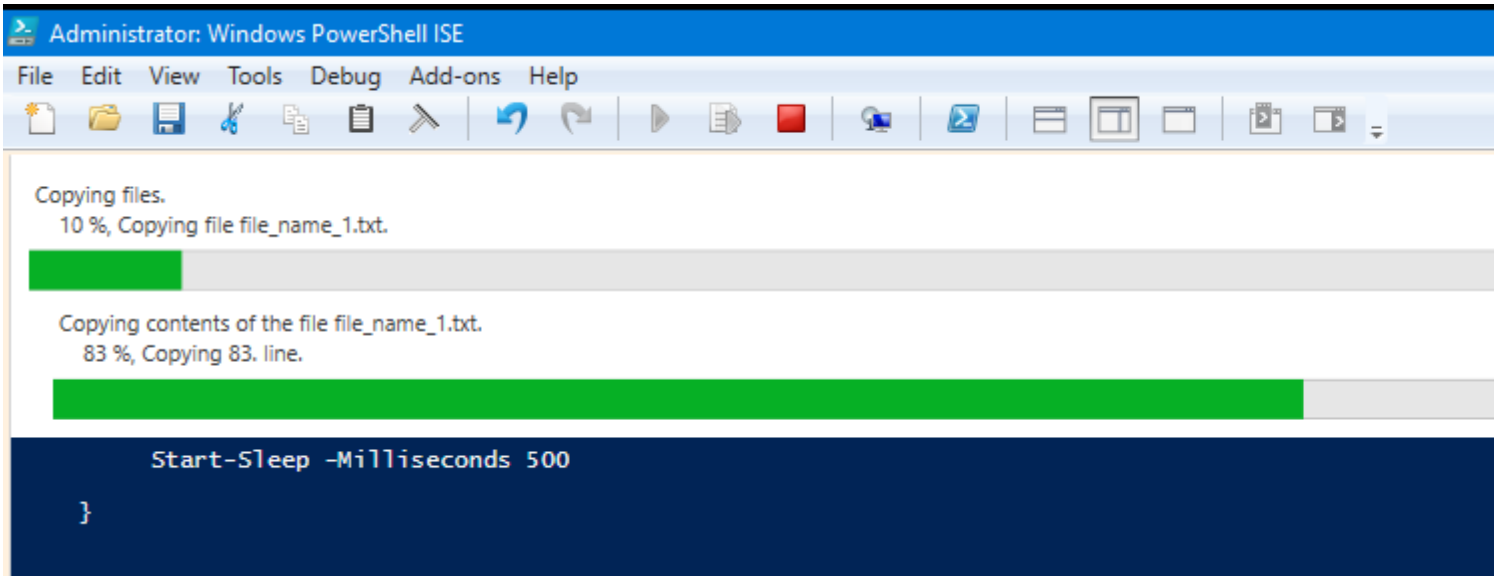
```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete
($_*10) -CurrentOperation "Copying file $fileName"

        1..100 | foreach-object {
            Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -
Id 2 -ParentId 1 -PercentComplete $_ -CurrentOperation "Copying $_. line"

            Start-Sleep -Milliseconds 20 # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
        }

        Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line with
your executive code (i.e. file search)

   }
```

Bitte beachten Sie, dass dieses Beispiel der Kürze halber keinen Code enthält (mit `Start-Sleep` simuliert). Es ist jedoch möglich, es direkt als solches auszuführen und dann zu modifizieren und damit zu spielen.

So sieht das Ergebnis in der PS-Konsole aus:



So sieht das Ergebnis in PS ISE aus:



Fortschrittsbalken verwenden online lesen:
https://riptutorial.com/de/powershell/topic/5020/fortschrittsbalken-verwenden

Kapitel 26: Gemeinsame Parameter

Bemerkungen

Gemeinsame Parameter können mit jedem Cmdlet verwendet werden (dh, sobald Sie Ihre Funktion als Cmdlet markieren [siehe `CmdletBinding()`], erhalten Sie alle diese Parameter kostenlos).

Hier ist die Liste aller gängigen Parameter (Alias steht nach dem entsprechenden Parameter in Klammern):

```
-Debug (db)
-ErrorAction (ea)
-ErrorVariable (ev)
-InformationAction (ia) # introduced in v5
-InformationVariable (iv) # introduced in v5
-OutVariable (ov)
-OutBuffer (ob)
-PipelineVariable (pv)
-Verbose (vb)
-WarningAction (wa)
-WarningVariable (wv)
-WhatIf (wi)
-Confirm (cf)
```

Examples

ErrorAction-Parameter

Mögliche Werte sind `Continue | Ignore | Inquire | SilentlyContinue | Stop | Suspend`

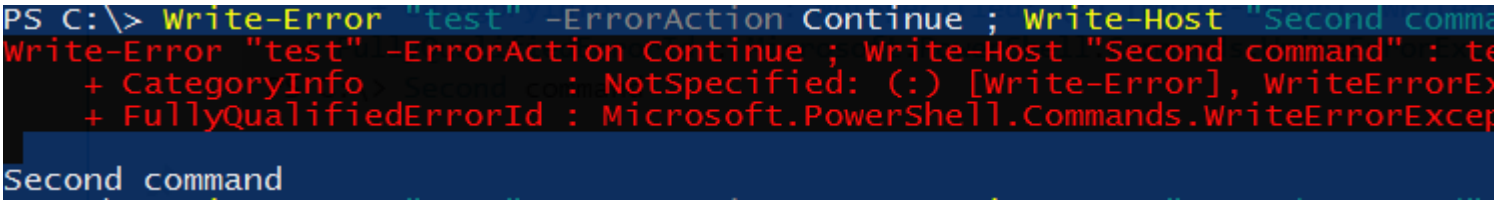
Der Wert dieses Parameters bestimmt, wie das Cmdlet nicht beendende Fehler behandelt (z. B. von Write-Error generierte Informationen; weitere Informationen zur Fehlerbehandlung finden Sie unter [*Noch nicht erstellter* Abschnitt]).

Der Standardwert (wenn dieser Parameter nicht angegeben wird) ist `Continue` .

-ErrorAction Weiter

Diese Option erzeugt eine Fehlermeldung und setzt die Ausführung fort.

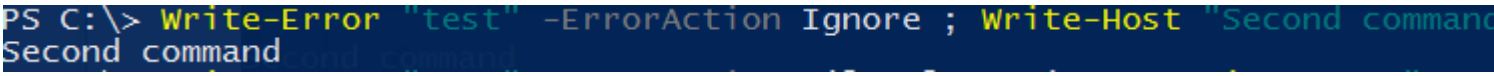
```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```



-ErrorAction Ignorieren

Diese Option erzeugt keine Fehlermeldung und setzt die Ausführung fort. Es werden auch keine Fehler zu der automatischen Variable `$Error` hinzugefügt.
Diese Option wurde in Version 3 eingeführt.

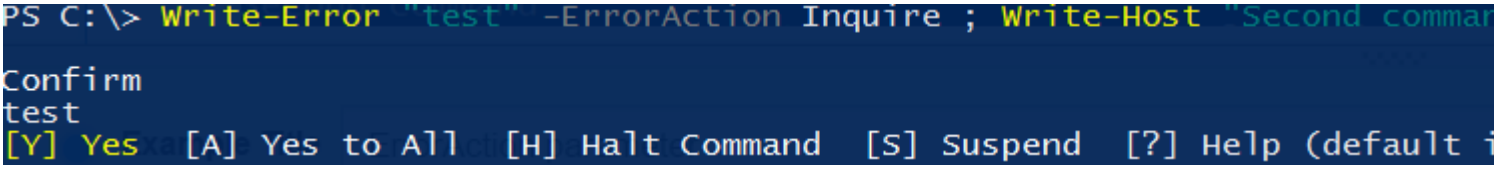
```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```



-ErrorAction-Anfrage

Diese Option erzeugt eine Fehlermeldung und fordert den Benutzer auf, eine auszuführende Aktion auszuwählen.

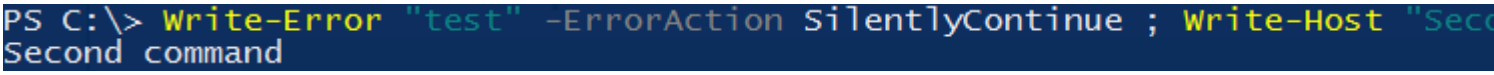
```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```



-ErrorAction SilentlyContinue

Diese Option erzeugt keine Fehlermeldung und setzt die Ausführung fort. Alle Fehler werden der automatischen Variable `$Error` hinzugefügt.

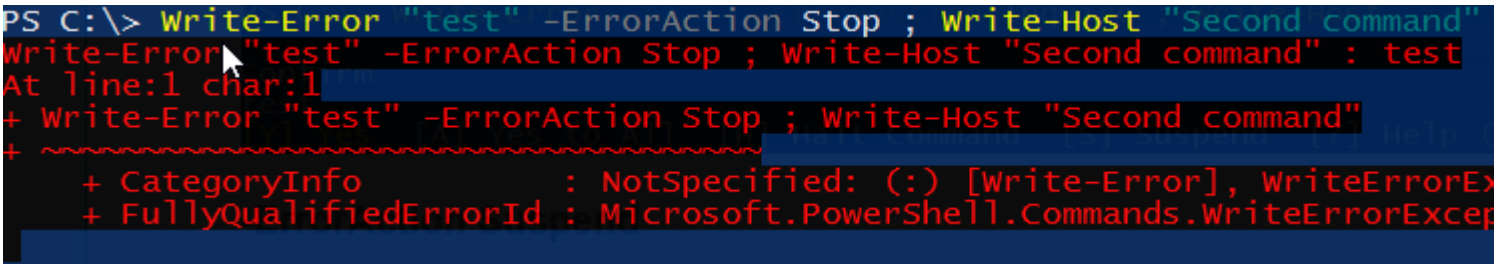
```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```



-ErrorAction Stop

Diese Option erzeugt eine Fehlermeldung und setzt die Ausführung nicht fort.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```



-ErrorAction Suspend

Nur in PowerShell-Workflows verfügbar. Wenn der Befehl verwendet wird und ein Fehler auftritt, wird der Workflow angehalten. Dies ermöglicht die Untersuchung eines solchen Fehlers und gibt die Möglichkeit, den Workflow wieder aufzunehmen. Weitere Informationen zum Workflow-System finden Sie unter [Thema wurde noch nicht erstellt].

Gemeinsame Parameter online lesen:
https://riptutorial.com/de/powershell/topic/5951/gemeinsame-parameter

Kapitel 27: Geplantes Aufgabenmodul

Einführung

Beispiele für die Verwendung des in Windows 8 / Server 2012 und höher verfügbaren Scheduled Tasks-Moduls.

Examples

Führen Sie das PowerShell-Skript als geplante Aufgabe aus

Erstellt eine geplante Aufgabe, die sofort ausgeführt wird. Anschließend wird `C:\myscript.ps1` als `SYSTEM`

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType
ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

Geplantes Aufgabenmodul online lesen:
https://riptutorial.com/de/powershell/topic/10940/geplantes-aufgabenmodul

Kapitel 28: Gewünschter Zustand Konfiguration

Examples

Einfaches Beispiel - Aktivieren von WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = “Present”
            Name = “Web-Server”
        }
    }
}
```

Wenn Sie diese Konfiguration in Powershell (EnableIISFeature) ausführen, wird eine Datei localhost.mof erstellt. Dies ist die "kompilierte" Konfiguration, die Sie auf einem Computer ausführen können.

Um die DSC-Konfiguration auf Ihrem localhost zu testen, können Sie einfach Folgendes aufrufen:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Starten von DSC (mof) auf einem Remote-Computer

Das Starten eines DSC auf einem Remote-Computer ist fast genauso einfach. Vorausgesetzt, Sie haben Powershell-Remoting bereits eingerichtet (oder WSMAN aktiviert).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Hinweis: Angenommen, Sie haben eine Konfiguration für Ihren Knoten auf Ihrer localmachine kompiliert (und dass die Datei myserver.somedomain.com.mof vor dem Starten der Konfiguration vorhanden ist).

Importieren von psd1 (Datendatei) in lokale Variable

Manchmal kann es nützlich sein, Ihre Powershell-Datendateien zu testen und die Knoten und Server zu durchlaufen.

Powershell 5 (WMF5) fügte diese nette kleine Funktion hinzu, genannt Import-PowerShellDataFile.

Beispiel:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Liste der verfügbaren DSC-Ressourcen

So listen Sie verfügbare DSC-Ressourcen auf Ihrem Authoring-Knoten auf:

```
 Get-DscResource
```

Dadurch werden alle Ressourcen für alle installierten Module (die sich in Ihrem PSModulePath befinden) auf Ihrem Authoring-Knoten aufgelistet.

Um alle verfügbaren DSC-Ressourcen aufzulisten, die in den Online-Quellen (PSGallery ++) auf WMF 5 verfügbar sind:

```
Find-DSCResource
```

Ressourcen für die Verwendung in DSC importieren

Bevor Sie eine Ressource in einer Konfiguration verwenden können, müssen Sie sie explizit importieren. Durch die Installation auf Ihrem Computer können Sie die Ressource nicht implizit verwenden.

Importieren Sie eine Ressource mit Import-DscResource.

Beispiel zum Importieren der PSDesiredStateConfiguration-Ressource und der File-Ressource.

```
Configuration InstallPreReqs
{
   param(); # params to DSC goes here.

   Import-DscResource PSDesiredStateConfiguration

   File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
 }
```

Hinweis : Damit DSC-Ressourcen funktionieren, müssen die Module auf den Zielcomputern installiert sein, wenn die Konfiguration ausgeführt wird. Wenn Sie sie nicht installiert haben, schlägt die Konfiguration fehl.

Gewünschter Zustand Konfiguration online lesen:
https://riptutorial.com/de/powershell/topic/5662/gewunschter-zustand-konfiguration

Kapitel 29: Grundlegende Vorgänge

Einführung

Ein Set ist eine Sammlung von Elementen, die alles sein können. Unabhängig von dem Operator, den wir für die Bearbeitung dieser Sets benötigen, handelt es sich um die *Set-Operatoren.* Die Operation wird auch als *Set-Operation bezeichnet* . Grundlegende Set-Operationen umfassen Union, Schnittpunkt sowie Addition, Subtraktion usw.

Syntax

- Gruppenobjekt
- Group-Object -Property <Eigenschaftsname>
- Group-Object -Property <propertyName>, <propertyName2>
- Group-Object -Property <propertyName> -CaseSensitive
- Gruppenobjekt -Eigenschaft <Eigenschaftsname> -Kultur <Kultur>
- Group-Object -Property <ScriptBlock>
- Sortierobjekt
- Sort-Object -Property <Eigenschaftsname>
- Sort-Object -Property <ScriptBlock>
- Sort-Object -Property <propertyName>, <propertyName2>
- Sort-Object -Property <propertyObject> -CaseSensitive
- Sort-Object -Property <propertyObject> -Descending
- Sort-Object -Property <propertyObject> -Unique
- Sort-Object -Property <propertyObject> -Kultur <Kultur>

Examples

Filtern: Where-Object / Where /?

Filtern Sie eine Aufzählung mithilfe eines bedingten Ausdrucks

Synonyme:

```
Where-Object
where
?
```

Beispiel:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Kehrt zurück:

Aaron
Albert
Alphonse

Sortierung: Sortierobjekt / Sortierung

Sortieren Sie eine Aufzählung in aufsteigender oder absteigender Reihenfolge

Synonyme:

```
Sort-Object
sort
```

Vorausgesetzt:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

Aufsteigende Sortierung ist die Standardeinstellung:

```
$names | Sort-Object
$names | sort
```

Aaron
Aaron
Bernie
Charlie
Danny

Absteigende Reihenfolge anfordern:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

Danny
Charlie

Bernie
Aaron
Aaron

Sie können mit einem Ausdruck sortieren.

```
$names | Sort-Object { $_.length }
```

Aaron
Aaron
Danny
Bernie
Charlie

Gruppierung: Gruppenobjekt / Gruppe

Sie können eine Aufzählung basierend auf einem Ausdruck gruppieren.

Synonyme:

```
Group-Object
group
```

Beispiele:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Group-Object -Property Length
$names | group -Property Length
```

Antwort:

Anzahl	Name	Gruppe
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{Albert, Bernie}
1	8	{Alphonse}
1	7	{Charlie}

Projizieren: Objekt auswählen / auswählen

Durch das Projizieren einer Aufzählung können Sie bestimmte Elemente jedes Objekts extrahieren, alle Details extrahieren oder Werte für jedes Objekt berechnen

Synonyme:

```
Select-Object
select
```

Auswahl einer Teilmenge der Eigenschaften:

```
$dir = dir "C:\MyFolder"

$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

Name	Vollständiger Name	Attribute
Bilder	C: \ MeinOrdner \ Bilder	Verzeichnis
data.txt	C: \ MeinOrdner \ data.txt	Archiv
quelle.c	C: \ MeinOrdner \ source.c	Archiv

Das erste Element auswählen und seine Eigenschaften anzeigen:

```
$d | select -first 1 *
```

PSPath
PSParentPath
PSChildName
PSDrive
PSProvider
PSIsContainer
BaseName
Modus
Name
Elternteil
Existiert
Wurzel
Vollständiger Name
Erweiterung

Erschaffungszeit
CreationTimeUtc
LastAccessTime
LastAccessTimeUtc
LastWriteTime
LastWriteTimeUtc
Attribute

Grundlegende Vorgänge online lesen:
https://riptutorial.com/de/powershell/topic/1557/grundlegende-vorgange

Kapitel 30: GUI in Powershell

Examples

WPF-GUI für das Get-Service-Cmdlet

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="Auto"
    SizeToContent="WidthAndHeight"
    Title="Get-Service">
    <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Margin="10,10,0,10">ComputerName:</Label>
                <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
            </StackPanel>
            <DockPanel>
                <Button Name="ButtonGetService" Content="Get-Service" Margin="10"
Width="150px" IsEnabled="false"/>
                <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right"
Margin="10" Width="50px"/>
            </DockPanel>
        </StackPanel>
    </ScrollViewer >
</Window>
'

# Create the Window Object
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# TextChanged Event Handler for Input
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
    $ComputerName = $TextboxInput.Text
    $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# Click Event Handler for ButtonClose
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
    $Window.Close();
})

# Click Event Handler for ButtonGetService
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
    $ComputerName = $TextboxInput.text.Trim()
    try{
        Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
    }catch{
```

```
[System.Windows.MessageBox]::Show($_.exception.message,"Error",[System.Windows.MessageBoxButton]::OK,[S

    }
})

# Open the Window
$Window.ShowDialog() | Out-Null
```

Dadurch wird ein Dialogfenster erstellt, in dem der Benutzer einen Computernamen auswählen kann. Anschließend wird eine Tabelle der Dienste und deren Status auf diesem Computer angezeigt.
In diesem Beispiel wird WPF anstelle von Windows Forms verwendet.

GUI in Powershell online lesen: https://riptutorial.com/de/powershell/topic/7141/gui-in-powershell

Kapitel 31: HashTables

Einführung

Eine Hash-Tabelle ist eine Struktur, die Schlüssel zu Werten abbildet. Weitere Informationen finden Sie in der Hash-Tabelle .

Bemerkungen

Ein wichtiges Konzept, das auf Hash-Tabellen beruht, ist das Splatting . Es ist sehr nützlich, um eine große Anzahl von Anrufen mit sich wiederholenden Parametern durchzuführen.

Examples

Hash-Tabelle erstellen

Beispiel zum Erstellen einer leeren HashTable:

```
$hashTable = @{}
```

Beispiel zum Erstellen einer HashTable mit Daten:

```
$hashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}
```

Greifen Sie mit einem Schlüssel auf einen Hashtabellenwert zu.

Ein Beispiel für das Definieren einer Hash-Tabelle und den Zugriff auf einen Wert mit dem Schlüssel

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Key1
#output
Value1
```

Ein Beispiel für den Zugriff auf einen Schlüssel mit ungültigen Zeichen für einen Eigenschaftennamen:

```
$hashTable = @{
    'Key 1' = 'Value3'
```

```
    Key2 = 'Value4'
}
$hashTable.'Key 1'
#Output
Value3
```

Durchlaufen einer Hashtabelle

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
#Output
Key1 : Value1
Key2 : Value2
```

Fügen Sie einer vorhandenen Hashtabelle ein Schlüsselwertpaar hinzu

Ein Beispiel zum Hinzufügen eines Schlüssels "Key2" mit dem Wert "Value2" zur Hashtabelle mithilfe des Additionsoperators:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Ein Beispiel zum Hinzufügen eines Schlüssels "Key2" mit dem Wert "Value2" zur Hashtabelle mithilfe der Add-Methode:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Aufzählung durch Schlüssel und Schlüsselwertpaare

Aufzählung über Schlüssel

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # or
    $value = $var1.$key
}
```

Aufzählung durch Schlüsselwertpaare

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

Entfernen Sie ein Schlüsselwertpaar aus einer vorhandenen Hashtabelle

Ein Beispiel zum Entfernen eines Schlüssels "Key2" mit dem Wert "Value2" aus der Hashtabelle mit dem Operator remove:

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
```

HashTables online lesen: https://riptutorial.com/de/powershell/topic/8083/hashtables

Kapitel 32: Infrastrukturautomatisierung

Einführung

Durch die Automatisierung von Infrastructure Management Services wird der FTE-Wert reduziert, und es werden kumulativ bessere ROI mit mehreren Tools, Orchestratoren, Orchestrierungs-Engine, Skripts und einfacher Benutzeroberfläche erzielt

Examples

Einfaches Skript für den Black-Box-Integrationstest von Konsolenanwendungen

Dies ist ein einfaches Beispiel, wie Sie Tests für eine Konsolenanwendung automatisieren können, die mit der Standardeingabe und der Standardausgabe interagieren.

Die getestete Anwendung liest und summiert jede neue Zeile und liefert das Ergebnis, nachdem eine einzige weiße Linie bereitgestellt wurde. Das Power-Shell-Skript schreibt "bestanden", wenn die Ausgabe übereinstimmt.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Infrastrukturautomatisierung online lesen:
https://riptutorial.com/de/powershell/topic/10909/infrastrukturautomatisierung

Kapitel 33: ISE-Modul

Einführung

ISE (Integrated Scripting Environment) von Windows PowerShell ist eine Hostanwendung, mit der Sie Skripts und Module in einer grafischen und intuitiven Umgebung schreiben, ausführen und testen können. Zu den wichtigsten Funktionen in Windows PowerShell ISE gehören Syntaxfärbung, Ausfüllen von Registerkarten, Intellisense, visuelles Debugging, Unicode-Kompatibilität und kontextsensitive Hilfe.

Examples

Skripte testen

Die einfache und dennoch leistungsstarke Anwendung der ISE besteht darin, im oberen Abschnitt Code (mit intuitiver Syntaxfärbung) zu schreiben und den Code auszuführen, indem er einfach markiert und die F8-Taste gedrückt wird.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum

1..10 | Get-Sum

#output
55
```

ISE-Modul online lesen: https://riptutorial.com/de/powershell/topic/10954/ise-modul

Kapitel 34: Kommentarbasierte Hilfe

Einführung

PowerShell bietet einen Dokumentationsmechanismus, der als kommentarbasierte Hilfe bezeichnet wird. Es ermöglicht die Dokumentation von Skripten und Funktionen mit Code-Kommentaren. Kommentarbasierte Hilfe wird meistens in Kommentarblöcken geschrieben, die mehrere Hilfeschlüsselwörter enthalten. Hilfeschlüsselwörter beginnen mit Punkten und identifizieren Hilfeabschnitte, die durch Ausführen des Cmdlets `Get-Help` angezeigt werden.

Examples

Funktionskommentare-basierte Hilfe

```
<#

.SYNOPSIS
    Gets the content of an INI file.

.DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

.INPUTS
    System.String

.OUTPUTS
    System.Collections.Hashtable

.PARAMETER FilePath
    Specifies the path to the input INI file.

.EXAMPLE
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini
    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.

.LINK
    Out-IniFile

#>
function Get-IniContent
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNullOrEmpty()]
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]
        [System.String]$FilePath
    )

    # Initialize output hash table.
    $ini = @{}
```

```
    switch -regex -file $FilePath
    {
        "^\[(.+)\]$" # Section
        {
            $section = $matches[1]
            $ini[$section] = @{}
            $CommentCount = 0
        }
        "^(;.*)$" # Comment
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $value = $matches[1]
            $CommentCount = $CommentCount + 1
            $name = "Comment" + $CommentCount
            $ini[$section][$name] = $value
        }
        "(.+?)\s*=\s*(.*)" # Key
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $name,$value = $matches[1..2]
            $ini[$section][$name] = $value
        }
    }

    return $ini
}
```

Die obige Funktionsdokumentation kann durch Ausführen von `Get-Help -Name Get-IniContent -Full` :

```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.


SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]


DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.


PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String


OUTPUTS
    System.Collections.Hashtable


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>$IniContent = Get-IniContent -FilePath file.ini

    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.




RELATED LINKS
    Out-IniFile



PS C:\Scripts>
```

Beachten Sie, dass die Kommentar-basierten Schlüsselwörter mit a beginnen `.` passen Sie die `Get-Help` Ergebnisabschnitte an.

Kommentar-basierte Hilfe

```
<#

.SYNOPSIS
    Reads a CSV file and filters it.
```

```
.DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.

.PARAMETER Path
    Specifies the path of the CSV input file.

.INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.

.OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

.EXAMPLE
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
Param
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

Die obige Skriptdokumentation kann durch Ausführen von `Get-Help -Name ReadUsersCsv.ps1 -Full` angezeigt werden:

```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full

NAME
    C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS
    Reads a CSV file and filters it.


SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]


DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.


PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                    true
        Position?                    2
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.


OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe




RELATED LINKS



PS C:\Scripts>
```

Kommentarbasierte Hilfe online lesen:
https://riptutorial.com/de/powershell/topic/9530/kommentarbasierte-hilfe

Kapitel 35: Kommunikation mit RESTful-APIs

Einführung

REST steht für Representational State Transfer (Repräsentativer Zustandstransfer) (manchmal als "ReST" bezeichnet). Es basiert auf einem zustandslosen Client-Server-Protokoll, das zwischengespeichert werden kann, und es wird hauptsächlich das HTTP-Protokoll verwendet. Es wird hauptsächlich zum Erstellen von Web-Services verwendet, die leicht, wartbar und skalierbar sind. Ein auf REST basierender Dienst wird als RESTful-Dienst bezeichnet. Die dafür verwendeten APIs sind RESTful-APIs. In PowerShell wird `Invoke-RestMethod` verwendet, um damit umzugehen.

Examples

Verwenden Sie Slack.com Incoming Webhooks

Definieren Sie Ihre Nutzdaten, die für möglicherweise komplexere Daten gesendet werden sollen

```
$Payload = @{ text="test string"; username="testuser" }
```

Verwenden Sie das `ConvertTo-Json` Json und `Invoke-RestMethod` , um den Aufruf auszuführen

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
(ConvertTo-Json $Payload)
```

Nachricht an hipChat senden

```
$params = @{
    Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
    Body = @{
        color = 'yellow'
        message = "This is a test message!"
        notify = $false
        message_format = "text"
    } | ConvertTo-Json
    ContentType = 'application/json'
}

Invoke-RestMethod @params
```

Verwenden von REST mit PowerShell-Objekten zum Abrufen und Übertragen einzelner Daten

Rufen Sie Ihre REST-Daten ab und speichern Sie sie in einem PowerShell-Objekt:

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Ändern Sie Ihre Daten:

```
$Post.title = "New Title"
```

Setzen Sie die REST-Daten zurück

```
$Json = $Post | ConvertTo-Json
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -
ContentType 'application/json'
```

Verwenden von REST mit PowerShell-Objekten zum Abrufen und POST vieler Elemente

Rufen Sie Ihre REST-Daten ab und speichern Sie sie in einem PowerShell-Objekt:

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

Ändern Sie viele Elemente in Ihren Daten:

```
$Users[0].name = "John Smith"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "Jane Smith"
$Users[1].email = "Jane.Smith@example.com"
```

POST alle REST-Daten zurück:

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

Verwenden von REST mit PowerShell zum Löschen von Elementen

Identifizieren Sie das Element, das gelöscht werden soll, und löschen Sie es:

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Kommunikation mit RESTful-APIs online lesen:
https://riptutorial.com/de/powershell/topic/3869/kommunikation-mit-restful-apis

Kapitel 36: Mit Objekten arbeiten

Examples

Objekte aktualisieren

Eigenschaften hinzufügen

Wenn Sie einem vorhandenen Objekt Eigenschaften hinzufügen möchten, können Sie das Cmdlet Add-Member verwenden. Bei PSObjects werden Werte in einer Art "Noteneigenschaften" gespeichert.

```
$object = New-Object -TypeName PSObject -Property @{
        Name = $env:username
        ID = 12
        Address = $null
    }

Add-Member -InputObject $object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty

# Returns
PS> $Object
Name ID Address SomeNewProp
---- -- ------- -----------
nem  12         A value
```

Sie können mit Select-Object-Cmdlet auch Eigenschaften hinzufügen (sogenannte berechnete Eigenschaften):

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'Another value'}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- -- ------- ----------- -------------
nem  12         A value     Another value
```

Der obige Befehl kann folgendermaßen verkürzt werden:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Eigenschaften entfernen

Mit dem Select-Object-Cmdlet können Sie Eigenschaften aus einem Objekt entfernen:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address
```

```
# Returns
PS> $object
Name SomeNewProp SomeOtherProp
---- ----------- -------------
nem  A value     Another value
```

Ein neues Objekt erstellen

Im Gegensatz zu anderen Skriptsprachen sendet PowerShell Objekte durch die Pipeline. Dies bedeutet, dass es beim Senden von Daten von einem Befehl an einen anderen wichtig ist, Objekte erstellen, ändern und sammeln zu können.

Ein Objekt zu erstellen ist einfach. Bei den meisten von Ihnen erstellten Objekten handelt es sich um benutzerdefinierte Objekte in PowerShell. Der zu verwendende Typ ist PSObject. In PowerShell können Sie auch alle Objekte erstellen, die Sie in .NET erstellen könnten.

Hier ein Beispiel zum Erstellen neuer Objekte mit einigen Eigenschaften:

Option 1: Neues Objekt

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Sie können das Objekt in einer Variablen speichern, indem Sie dem Befehl `$newObject =`

Möglicherweise müssen Sie auch Sammlungen von Objekten speichern. Dazu können Sie eine leere Sammlungsvariable erstellen und der Sammlung Objekte hinzufügen.

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

Sie können dann dieses Sammlungsobjekt nach Objekt durchlaufen. Suchen Sie dazu den Loop-Abschnitt in der Dokumentation.

Option 2: Auswahlobjekt

Eine weniger gebräuchliche Methode zum Erstellen von Objekten, die Sie noch im Internet finden, ist die folgende:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Option 3: pscustomobject-Beschleuniger (PSv3 + erforderlich)

Der bestellte Beschleuniger zwingt PowerShell dazu, unsere Eigenschaften in der von uns definierten Reihenfolge zu halten. Sie benötigen den bestellten Typ-Beschleuniger nicht, um `[PSCustomObject]` :

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Objekt untersuchen

Nun, da Sie ein Objekt haben, kann es gut sein, herauszufinden, was es ist. Mit dem Cmdlet Get-Member können Sie sehen, was ein Objekt ist und was es enthält:

```
Get-Item c:\windows | Get-Member
```

Dies ergibt:

```
TypeName: System.IO.DirectoryInfo
```

Gefolgt von einer Liste von Eigenschaften und Methoden des Objekts.

Eine andere Möglichkeit, den Typ eines Objekts abzurufen, ist die Verwendung der GetType-Methode.

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryInfo                            System.IO.FileSystemInfo
```

Um eine Liste der Eigenschaften anzuzeigen, die das Objekt zusammen mit seinen Werten besitzt, können Sie das Cmdlet Format-List mit dem Parameter Property verwenden, der auf * gesetzt ist.

Hier ist ein Beispiel mit der resultierenden Ausgabe:

```
C:\> Get-Item C:\Windows | Format-List -Property *


PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName       : Windows
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-----
BaseName          : Windows
Target            : {}
LinkType          :
Name              : Windows
Parent            :
Exists            : True
Root              : C:\
FullName          : C:\Windows
Extension         :
CreationTime      : 30/10/2015 06:28:30
CreationTimeUtc   : 30/10/2015 06:28:30
LastAccessTime    : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime     : 16/08/2016 17:32:04
LastWriteTimeUtc  : 16/08/2016 16:32:04
Attributes        : Directory
```

Instanzen generischer Klassen erstellen

Hinweis: Für PowerShell 5.1 geschriebene Beispiele Sie können Instanzen generischer Klassen erstellen

```
#Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#Normal System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
```

```
$aDate = $null #Throws exception when PowerShell attempts to convert null to
```

Gibt die Ausgabe aus:

```
Saturday, 4 August 2012 08:53:02
System.DateTime
Sunday, 4 August 2013 08:53:02
System.DateTime
Cannot convert null to type "System.DateTime".
At line:14 char:1
+ $aDate = $null
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Generische Sammlungen sind ebenfalls möglich

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')


$dict.Add('4', 'd') #powershell auto converts '4' to 4
$dict.Add('5.1', 'c') #powershell auto converts '5.1' to 5

$dict

$dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so it throws an error
```

Gibt die Ausgabe aus:

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

Key Value
--- -----
  1 a
  2 b
  3 c
  4 d
  5 c
Cannot convert argument "key", with value: "z", for "Add" to type "System.Int32": "Cannot
convert value "z" to type "System.Int32". Error: "Input string was not in a correct format.""
At line:15 char:1
+ $dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so  ...
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

Mit Objekten arbeiten online lesen: https://riptutorial.com/de/powershell/topic/1328/mit-objekten-arbeiten

Kapitel 37: Mit XML-Dateien arbeiten

Examples

Zugriff auf eine XML-Datei

```
<!-- file.xml -->
<people>
    <person id="101">
        <name>Jon Lajoie</name>
        <age>22</age>
    </person>
    <person id="102">
        <name>Lord Gaben</name>
        <age>65</age>
    </person>
    <person id="103">
        <name>Gordon Freeman</name>
        <age>29</age>
    </person>
</people>
```

Laden einer XML-Datei

Zum Laden einer XML-Datei können Sie eine der folgenden verwenden:

```
# First Method
$xdoc = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xdoc.load($file)

# Second Method
[xml] $xdoc = Get-Content ".\file.xml"

# Third Method
$xdoc = [xml] (Get-Content ".\file.xml")
```

Zugriff auf XML als Objekte

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.people.person

id                                      name                                    age
--                                      ----                                    ---
```

```
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.people.person[0].name
Jon Lajoie

PS C:\> $xml.people.person[1].age
65

PS C:\> $xml.people.person[2].id
103
```

Zugriff auf XML mit XPath

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.SelectNodes("//people//person")

id                                      name                                    age
--                                      ----                                    ---
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
Jon Lajoie

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103
```

Zugriff auf XML mit Namespaces mit XPath

```
PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
    <ns:person id="101">
        <ns:name>Jon Lajoie</ns:name>
    </ns:person>
    <ns:person id="102">
        <ns:name>Lord Gaben</ns:name>
    </ns:person>
    <ns:person id="103">
        <ns:name>Gordon Freeman</ns:name>
    </ns:person>
</ns:people>
"@

PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
```

```
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)

id                                   name
--                                   ----
101                                  Jon Lajoie
102                                  Lord Gaben
103                                  Gordon Freeman
```

XML-Dokument mit XmlWriter () erstellen

```
# Set The Formatting
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# Set the File Name Create The Document
$XmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Write the XML Decleration and set the XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Start the Root Element
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- Start <Object>

        $xmlWriter.WriteElementString("Property1","Value 1")
        $xmlWriter.WriteElementString("Property2","Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
            $xmlWriter.WriteElementString("Property3","Value 3")
        $xmlWriter.WriteEndElement() # <-- End <SubObject>

    $xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# End, Finalize and close the XML Document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

XML-Datei ausgeben

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
    <Object>
        <Property1>Value 1</Property1>
        <Property2>Value 2</Property2>
        <SubObject>
            <Property3>Value 3</Property3>
        </SubObject>
    </Object>
</Root>
```

Beispieldaten

XML-Dokument

Zunächst definieren wir ein XML-Beispieldokument mit dem Namen " **books.xml** " in unserem aktuellen Verzeichnis:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>Of Mice And Men</title>
        <author>John Steinbeck</author>
        <pageCount>187</pageCount>
        <publishers>
            <publisher>
                <isbn>978-88-58702-15-4</isbn>
                <name>Pascal Covici</name>
                <year>1937</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-05-82461-46-8</isbn>
                <name>Longman</name>
                <year>2009</year>
                <binding>Hardcover</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Lennie Small" />
            <character name="Curley's Wife" />
            <character name="George Milton" />
            <character name="Curley" />
        </characters>
        <film>True</film>
    </book>
    <book>
        <title>The Hunt for Red October</title>
        <author>Tom Clancy</author>
        <pageCount>387</pageCount>
        <publishers>
            <publisher>
                <isbn>978-08-70212-85-7</isbn>
                <name>Naval Institute Press</name>
                <year>1984</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-04-25083-83-3</isbn>
                <name>Berkley</name>
                <year>1986</year>
                <binding>Paperback</binding>
            </publisher>
```

```
            <publisher>
                <isbn>978-08-08587-35-4</isbn>
                <name>Penguin Putnam</name>
                <year>2010</year>
                <binding>Paperback</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Marko Alexadrovich Ramius" />
            <character name="Jack Ryan" />
            <character name="Admiral Greer" />
            <character name="Bart Mancuso" />
            <character name="Vasily Borodin" />
        </characters>
        <film>True</film>
    </book>
</books>
```

Neue Daten

Was wir tun wollen, ist, diesem Dokument ein paar neue Bücher hinzuzufügen, sagen wir *Patriot Games* von Tom Clancy (ja, ich bin ein Fan von Clancy's Werken) und ein Sci-Fi-Favorit: *der Anhalter durch die Galaxis* von Douglas Adams, hauptsächlich weil Zaphod Beeblebrox einfach Spaß macht, zu lesen.

Irgendwie haben wir die Daten für die neuen Bücher erfasst und als Liste von PSCustomObjects gespeichert:

```
$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "Year" = "1979";
                "First" = $True;
```

```
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
            "Trillian", "Slartibartfast", "Dirk Gently"
        );
        "film" = $True;
    }
);
```

Vorlagen

Nun müssen wir einige grundlegende XML-Strukturen definieren, in die unsere neuen Daten einfließen können. Grundsätzlich möchten Sie für jede Datenliste ein Skelett / eine Vorlage erstellen. In unserem Beispiel bedeutet dies, dass wir eine Vorlage für das Buch, die Charaktere und die Verleger benötigen. Wir können dies auch nutzen einige Standardwerte, wie der Wert für den definieren `film` - Tag.

```
$t_book = [xml] @'
<book>
    <title />
    <author />
    <pageCount />
    <publishers />
    <characters />
    <film>False</film>
</book>
'@;

$t_publisher = [xml] @'
<publisher>
    <isbn/>
    <name/>
    <year/>
    <binding/>
    <first>false</first>
</publisher>
'@;

$t_character = [xml] @'
<character name="" />
'@;
```

Wir sind mit der Einrichtung fertig.

Neue Daten hinzufügen

Nachdem wir nun alle Beispieldaten eingerichtet haben, fügen wir die benutzerdefinierten Objekte zum XML-Dokumentobjekt hinzu.

```
# Read the xml document
$xml = [xml] Get-Content .\books.xml;

# Let's show a list of titles to see what we've got currently:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};;

# Outputs:
# title                               author         ISBN
# -----                               ------         ----
# Of Mice And Men                     John Steinbeck 978-88-58702-15-4
# The Hunt for Red October            Tom Clancy     978-08-70212-85-7

# Let's show our new books as well:
$newBooks | Select Title, Author, @{N="ISBN";E={$_.Publishers[0].ISBN}};

# Outputs:
# Title                               Author        ISBN
# -----                               ------        ----
# Patriot Games                       Tom Clancy    978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# Now to merge the two:

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # Add the template for a book as a new node to the root element
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # Select the new child element
    $newElement = $root.SelectSingleNode("book[last()]");

    # Update the parameters of that new element to match our current new book data
    $newElement.title     = [String]$book.Title;
    $newElement.author    = [String]$book.Author;
    $newElement.pageCount = [String]$book.PageCount;
    $newElement.film      = [String]$book.Film;

    # Iterate through the properties that are Children of our new Element:
    ForEach ( $publisher in $book.Publishers ) {
        # Create the new child publisher element
        # Note the use of "SelectSingleNode" here, this allows the use of the "AppendChild"
method as it returns
        # a XmlElement type object instead of the $Null data that is currently stored in that
leaf of the
        # XML document tree

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher,
$true));

        # Update the attribute and text values of our new XML Element to match our new data
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        If ( $publisher.first ) {
            $newPublisherElement.first = "True";
        }
    }
```

```
    ForEach ( $character in $book.Characters ) {
        # Select the characters xml element
        $charactersElement = $newElement.SelectSingleNode("characters");

        # Add a new character child element
        [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

        # Select the new characters/character element
        $characterElement = $charactersElement.SelectSingleNode("character[last()]");

        # Update the attribute and text values to match our new data
        $characterElement.name = [String]$character;
    }
}

# Check out the new XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                              author         ISBN
# -----                              ------         ----
# Of Mice And Men                    John Steinbeck 978-88-58702-15-4
# The Hunt for Red October           Tom Clancy     978-08-70212-85-7
# Patriot Games                      Tom Clancy     978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams  978-0-33-025864-7
```

Wir können jetzt unser XML auf Festplatte, Bildschirm, Internet oder wo auch immer schreiben!

Profitieren

Dies ist zwar nicht für jeden die Prozedur, aber ich habe es gefunden, um eine Menge `[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementN` gefolgt von `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

Ich denke, die im Beispiel beschriebene Methode ist sauberer und für normale Menschen leichter zu analysieren.

Verbesserungen

Es kann möglich sein, die Vorlage so zu ändern, dass sie Elemente mit untergeordneten Elementen enthält, anstatt jeden Abschnitt als separate Vorlage aufzubrechen. Sie müssen nur darauf achten, das vorherige Element zu klonen, wenn Sie die Liste durchlaufen.

Mit XML-Dateien arbeiten online lesen: https://riptutorial.com/de/powershell/topic/4882/mit-xml-dateien-arbeiten

Kapitel 38: Module, Skripte und Funktionen

Einführung

PowerShell-Module bieten Erweiterungen für Systemadministrator, DBA und Entwickler. Ob es sich einfach um eine Methode handelt, um Funktionen und Skripte gemeinsam zu nutzen.

Powershell-Funktionen dienen zur Vermeidung von Wiederholungscodes. Siehe [PS-Funktionen] [1] [1]: PowerShell-Funktionen

PowerShell-Skripts werden für die Automatisierung von Verwaltungsaufgaben verwendet, die aus einer Befehlszeilen-Shell und zugehörigen, auf .NET Framework aufgebauten Cmdlets bestehen.

Examples

Funktion

Eine Funktion ist ein benannter Codeblock, mit dem wiederverwendbarer Code definiert wird, der einfach zu verwenden ist. Es ist normalerweise in einem Skript enthalten, um die Wiederverwendung von Code zu erleichtern (um doppelten Code zu vermeiden) oder als Teil eines Moduls verteilt zu werden, damit er für andere in mehreren Skripts nützlich ist.

Szenarien, in denen eine Funktion nützlich sein kann:

- Berechnen Sie den Durchschnitt einer Zahlengruppe
- Generieren Sie einen Bericht für laufende Prozesse
- Schreiben Sie eine Funktion, die testet, ob ein Computer "in Ordnung" ist, indem Sie einen Ping-Befehl an den Computer senden und auf die `c$` -share zugreifen

Funktionen werden mit dem `function` , gefolgt von einem einzelnen Wortnamen und einem Skriptblock, der den Code enthält, der beim Aufruf des Funktionsnamens ausgeführt werden soll.

```
function NameOfFunction {
    Your code
}
```

Demo

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

Verwendungszweck:

```
> HelloWorld
Greetings from PowerShell!
```

Skript

Ein Skript ist eine Textdatei mit der Dateierweiterung `.ps1` , die PowerShell-Befehle enthält, die beim Aufruf des Skripts ausgeführt werden. Da es sich bei Skripts um gespeicherte Dateien handelt, können sie problemlos zwischen Computern übertragen werden.

Skripts werden häufig geschrieben, um ein bestimmtes Problem zu lösen, z. B .:

- Führen Sie eine wöchentliche Wartungsaufgabe aus
- So installieren und konfigurieren Sie eine Lösung / Anwendung auf einem Computer

Demo

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

Sie können ein Skript ausführen, indem Sie den Pfad zur Datei mit folgendem Befehl eingeben:

- Absoluter Pfad, z. `c:\MyFirstScript.ps1`
- Relativer Pfad, z `.\MyFirstScript.ps1` wenn das aktuelle Verzeichnis der PowerShell-Konsole `C:\`

Verwendungszweck:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Ein Skript kann auch Module importieren, eigene Funktionen definieren usw.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Verwendungszweck:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
Let's get started!
4
Greetings from PowerShell!
```

Modul

Ein Modul ist eine Sammlung verwandter wiederverwendbarer Funktionen (oder Cmdlets), die problemlos an andere PowerShell-Benutzer verteilt und in mehreren Skripts oder direkt in der Konsole verwendet werden können. Ein Modul wird normalerweise in einem eigenen Verzeichnis gespeichert und besteht aus:

- Eine oder mehrere `.psm1` mit der Erweiterung `.psm1` , die Funktionen enthalten, oder binäre Assemblys (`.dll`), die Cmdlets enthalten
- Ein `.psd1` , das den `.psd1` , die Version, den Autor, die Beschreibung, die von ihm `.psd1` Funktionen / Cmdlets usw. beschreibt.
- Andere Voraussetzungen für die Arbeit inkl. Abhängigkeiten, Skripte usw.

Beispiele für Module:

- Ein Modul mit Funktionen / Cmdlets, die Statistiken für eine Datenmenge ausführen
- Ein Modul zum Abfragen und Konfigurieren von Datenbanken

Damit PowerShell ein Modul leicht finden und importieren kann, befindet es sich häufig an einem der bekannten Speicherorte für PowerShell-Module, die in `$env:PSModulePath` .

Demo

Listen Sie Module auf, die an einem der bekannten Speicherorte des Moduls installiert sind:

```
Get-Module -ListAvailable
```

Ein Modul importieren, z. `Hyper-V` Modul:

```
Import-Module Hyper-V
```

Liste der verfügbaren Befehle in einem Modul, z. das `Microsoft.PowerShell.Archive` -Modul

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name             Version Source
----------- ----             ------- ------
Function    Compress-Archive 1.0.1.0 Microsoft.PowerShell.Archive
Function    Expand-Archive   1.0.1.0 Microsoft.PowerShell.Archive
```

Erweiterte Funktionen

Erweiterte Funktionen verhalten sich wie Cmdlets. Die PowerShell ISE umfasst zwei Skelette mit erweiterten Funktionen. Rufen Sie diese über das Menü, Bearbeiten, Codeausschnitte oder durch Drücken von Strg + J auf. (Ab PS 3.0 können spätere Versionen abweichen)

Zu den wichtigsten Funktionen der erweiterten Funktionen gehören:

- integrierte, angepasste Hilfe für die Funktion, auf die über `Get-Help`
- kann [CmdletBinding ()] verwenden, wodurch die Funktion wie ein Cmdlet wirkt
- umfangreiche Parameteroptionen

Einfache Version:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Param2 help description
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Komplette Version:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
```

```
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipeline=$true,
                   ValueFromPipelineByPropertyName=$true,
                   ValueFromRemainingArguments=$false,
                   Position=0,
                   ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
```

```
    End
    {
    }
}
```

Module, Skripte und Funktionen online lesen:
https://riptutorial.com/de/powershell/topic/5755/module--skripte-und-funktionen

Kapitel 39: MongoDB

Bemerkungen

Am schwierigsten ist es, ein **Filialdokument** in das Dokument **einzufügen,** das noch nicht erstellt wurde. Wenn das Filialdokument in der erwarteten Form sein muss, müssen wir das Array mit einer for-Schleife in eine Variable `$doc2.add("Key", "Value")` und `$doc2.add("Key", "Value")` stattdessen das aktuelle Array von `foreach` mit Index verwenden. Dadurch wird das Filialdokument in zwei Zeilen aufgeteilt, wie Sie in den `"Tags" = [MongoDB.Bson.BsonDocument] $doc2` .

Examples

MongoDB mit C # -Treiber 1.7 mit PowerShell

Ich muss alle Details von der virtuellen Maschine abfragen und die MongoDB aktualisieren.

```
Which require the output look like this.
{
    "_id" : ObjectId("5800509f23888a12bccf2347"),
    "ResourceGrp" : "XYZZ-MachineGrp",
    "ProcessTime" : ISODate("2016-10-14T03:27:16.586Z"),
    "SubscriptionName" : "GSS",
    "OS" : "Windows",
    "HostName" : "VM1",
    "IPAddress" : "192.168.22.11",
    "Tags" : {
        "costCenter" : "803344",
        "BusinessUNIT" : "WinEng",
        "MachineRole" : "App",
        "OwnerEmail" : "zteffer@somewhere.com",
        "appSupporter" : "Steve",
        "environment" : "Prod",
        "implementationOwner" : "xyzr@somewhere.com",
        "appSoftware" : "WebServer",
        "Code" : "Gx",
        "WholeOwner" : "zzzgg@somewhere.com"
    },
    "SubscriptionID" : "",
    "Status" : "running fine",
    "ResourceGroupName" : "XYZZ-MachineGrp",
    "LocalTime" : "14-10-2016-11:27"
}
```

Ich habe 3 Sätze von Arrays in Powershell

```
        $MachinesList  # Array
        $ResourceList # Array
        $MachineTags  # Array

    pseudo code
```

```
        $mongoDriverPath = 'C:\Program Files (x86)\MongoDB\CSharpDriver 1.7';
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Bson.dll";
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Driver.dll";

$db = [MongoDB.Driver.MongoDatabase]::Create('mongodb://127.0.0.1:2701/RGrpMachines');
   [System.Collections.ArrayList]$TagList = $vm.tags
     $A1 = $Taglist.key
     $A2 = $Taglist.value
   foreach ($Machine in $MachinesList)
   {
       foreach($Resource in $ResourceList)
       {
                   $doc2 = $null
                  [MongoDB.Bson.BsonDocument] $doc2 = @{}; #Create a Document here
                   for($i = 0; $i -lt $TagList.count; $i++)
                          {
                                $A1Key = $A1[$i].ToString()
                                $A2Value = $A2[$i].toString()
                                $doc2.add("$A1Key", "$A2Value")
                          }

                    [MongoDB.Bson.BsonDocument] $doc = @{
                       "_id"= [MongoDB.Bson.ObjectId]::GenerateNewId();
                       "ProcessTime"= [MongoDB.Bson.BsonDateTime] $ProcessTime;
                       "LocalTime" = "$LocalTime";
                       "Tags" = [MongoDB.Bson.BsonDocument] $doc2;
                       "ResourceGrp" = "$RGName";
                       "HostName"= "$VMName";
                       "Status"= "$VMStatus";
                       "IPAddress"= "$IPAddress";
                       "ResourceGroupName"= "$RGName";
                       "SubscriptionName"= "$CurSubName";
                       "SubscriptionID"= "$subid";
                       "OS"= "$OSType";
                   }; #doc loop close

                   $collection.Insert($doc);
           }
   }
```

MongoDB online lesen: https://riptutorial.com/de/powershell/topic/7438/mongodb

Kapitel 40: Operatoren

Einführung

Ein Operator ist ein Zeichen, das eine Aktion darstellt. Sie weist den Compiler / Interpreter an, bestimmte mathematische, relationale oder logische Operationen auszuführen und das Endergebnis zu erzeugen. PowerShell interpretiert auf eine bestimmte Art und Weise und kategorisiert dementsprechend wie arithmetische Operatoren Operationen hauptsächlich mit Zahlen, sie wirken sich jedoch auch auf Zeichenfolgen und andere Datentypen aus. Neben den Basisoperatoren verfügt PowerShell über eine Reihe von Operatoren, die Zeit und Codieraufwand sparen (z. B. -like, -match, -replace usw.).

Examples

Rechenzeichen

```
1 + 2      # Addition
1 - 2      # Subtraction
-1         # Set negative value
1 * 2      # Multiplication
1 / 2      # Division
1 % 2      # Modulus
100 -shl 2 # Bitwise Shift-left
100 -shr 1 # Bitwise Shift-right
```

Logische Operatoren

```
-and # Logical and
-or  # Logical or
-xor # Logical exclusive or
-not # Logical not
!    # Logical not
```

Zuweisungsoperatoren

Einfache arithmetik:

```
$var = 1      # Assignment. Sets the value of a variable to the specified value
$var += 2     # Addition. Increases the value of a variable by the specified value
$var -= 1     # Subtraction. Decreases the value of a variable by the specified value
$var *= 2     # Multiplication. Multiplies the value of a variable by the specified value
$var /= 2     # Division. Divides the value of a variable by the specified value
$var %= 2     # Modulus. Divides the value of a variable by the specified value and then
              # assigns the remainder (modulus) to the variable
```

Inkrement und Dekrement:

```
$var++   # Increases the value of a variable, assignable property, or array element by 1
$var--   # Decreases the value of a variable, assignable property, or array element by 1
```

Vergleichsoperatoren

PowerShell-Vergleichsoperatoren bestehen aus einem führenden Bindestrich (`-`), gefolgt von einem Namen (`eq` für `equal` , `gt` für `greater than` , usw.).

Namen können mit Sonderzeichen vorangestellt werden, um das Verhalten des Operators zu ändern:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Die Groß- und Kleinschreibung wird nicht berücksichtigt, wenn nicht angegeben ("a" -eq "A") und ("a" -ieq "A").

Einfache Vergleichsoperatoren:

```
2 -eq 2    # Equal to (==)
2 -ne 4    # Not equal to (!=)
5 -gt 2    # Greater-than (>)
5 -ge 5    # Greater-than or equal to (>=)
5 -lt 10   # Less-than (<)
5 -le 5    # Less-than or equal to (<=)
```

String-Vergleichsoperatoren:

```
"MyString" -like "*String"           # Match using the wildcard character (*)
"MyString" -notlike "Other*"         # Does not match using the wildcard character (*)
"MyString" -match "$String^"         # Matches a string using regular expressions
"MyString" -notmatch "$Other^"       # Does not match a string using regular expressions
```

Vergleichsoperatoren für Sammlungen:

```
"abc", "def" -contains "def"           # Returns true when the value (right) is present
                                       # in the array (left)
"abc", "def" -notcontains "123"        # Returns true when the value (right) is not present
                                       # in the array (left)
"def" -in "abc", "def"                 # Returns true when the value (left) is present
                                       # in the array (right)
"123" -notin "abc", "def"              # Returns true when the value (left) is not present
                                       # in the array (right)
```

Umleitungsoperatoren

Erfolgsausgabestrom:

```
cmdlet > file     # Send success output to file, overwriting existing content
cmdlet >> file    # Send success output to file, appending to existing content
cmdlet 1>&2       # Send success and error output to error stream
```

Fehlerausgabestrom:

```
cmdlet 2> file    # Send error output to file, overwriting existing content
cmdlet 2>> file   # Send error output to file, appending to existing content
cmdlet 2>&1       # Send success and error output to success output stream
```

Warnungsausgabestrom: (PowerShell 3.0+)

```
cmdlet 3> file    # Send warning output to file, overwriting existing content
cmdlet 3>> file   # Send warning output to file, appending to existing content
cmdlet 3>&1       # Send success and warning output to success output stream
```

Ausführlicher Ausgabestrom: (PowerShell 3.0+)

```
cmdlet 4> file    # Send verbose output to file, overwriting existing content
cmdlet 4>> file   # Send verbose output to file, appending to existing content
cmdlet 4>&1       # Send success and verbose output to success output stream
```

Debug-Ausgabestream: (PowerShell 3.0 und höher)

```
cmdlet 5> file    # Send debug output to file, overwriting existing content
cmdlet 5>> file   # Send debug output to file, appending to existing content
cmdlet 5>&1       # Send success and debug output to success output stream
```

Informationsausgabestream: (PowerShell 5.0 und höher)

```
cmdlet 6> file    # Send information output to file, overwriting existing content
cmdlet 6>> file   # Send information output to file, appending to existing content
cmdlet 6>&1       # Send success and information output to success output stream
```

Alle Ausgabeströme:

```
cmdlet *> file    # Send all output streams to file, overwriting existing content
cmdlet *>> file   # Send all output streams to file, appending to existing content
cmdlet *>&1       # Send all output streams to success output stream
```

Unterschiede zum Pipe Operator (|)

Umleitungsoperatoren leiten nur Streams zu Dateien oder Streams zu Streams um. Der Pipe-Operator pumpt ein Objekt entlang der Pipeline zu einem Cmdlet oder der Ausgabe. Die Funktionsweise der Pipeline unterscheidet sich im Allgemeinen von der Funktionsweise der Umleitung und kann unter Arbeiten mit der PowerShell-Pipeline gelesen werden

Mischen von Operandentypen: Der Typ des linken Operanden bestimmt das Verhalten.

Für die Zugabe

```
"4" + 2         # Gives "42"
4 + "2"         # Gives 6
```

```
1,2,3 + "Hello" # Gives 1,2,3,"Hello"
"Hello" + 1,2,3 # Gives "Hello1 2 3"
```

Zur Multiplikation

```
"3" * 2   # Gives "33"
2 * "3"   # Gives 6
1,2,3 * 2 # Gives 1,2,3,1,2,3
2 * 1,2,3 # Gives an error op_Multiply is missing
```

Die Auswirkungen können versteckte Folgen für Vergleichsoperatoren haben:

```
$a = Read-Host "Enter a number"
Enter a number : 33
$a -gt 5
False
```

String-Manipulationsoperatoren

Operator ersetzen:

Der `-replace` Operator ersetzt ein Muster in einem Eingabewert mithilfe eines regulären Ausdrucks. Dieser Operator verwendet zwei Argumente (durch ein Komma getrennt): ein Muster für reguläre Ausdrücke und dessen Ersetzungswert (der optional ist und standardmäßig eine leere Zeichenfolge).

```
"The rain in Seattle" -replace 'rain','hail'        #Returns: The hail in Seattle
"kenmyer@contoso.com" -replace '^[\w]+@(.+)', '$1'  #Returns: contoso.com
```

Split- und Join-Operatoren:

Der `-split` Operator teilt einen String in ein Array von Sub-Strings auf.

```
"A B C" -split " "      #Returns an array string collection object containing A,B and C.
```

Der `-join` Operator `-join` ein String-Array zu einem einzigen String zusammen.

```
"E","F","G" -join ":"   #Returns a single string: E:F:G
```

Operatoren online lesen: https://riptutorial.com/de/powershell/topic/1071/operatoren

Kapitel 41: Paketverwaltung

Einführung

Mit PowerShell Package Management können Sie PowerShell-Module und andere Pakete finden, installieren, aktualisieren und deinstallieren.

PowerShellGallery.com ist die Standardquelle für PowerShell-Module. Sie können die Site auch nach verfügbaren Paketen durchsuchen, den Code anweisen und eine Vorschau anzeigen.

Examples

Suchen Sie ein PowerShell-Modul anhand eines Musters

So finden Sie ein Modul, das mit `DSC` endet

```
Find-Module -Name *DSC
```

Erstellen Sie das Standard-Reposity-PowerShell-Modul

Wenn das PowerShell-Standard-Repository `PSGallery` aus irgendeinem Grund entfernt wird, wird es entfernt. Sie müssen es erstellen. Dies ist der Befehl.

```
Register-PSRepository -Default
```

Suchen Sie ein Modul anhand des Namens

```
Find-Module -Name <Name>
```

Installieren Sie ein Modul anhand des Namens

```
Install-Module -Name <name>
```

Deinstallieren Sie ein Modul, meinen Namen und meine Version

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Aktualisieren Sie ein Modul anhand des Namens

```
Update-Module -Name <Name>
```

Paketverwaltung online lesen: https://riptutorial.com/de/powershell/topic/8698/paketverwaltung

Kapitel 42: Parametersätze

Einführung

Parametersätze werden verwendet, um die mögliche Kombination von Parametern einzuschränken oder die Verwendung von Parametern zu erzwingen, wenn ein oder mehrere Parameter ausgewählt werden.

Die Beispiele erläutern die Verwendung und den Grund eines Parametersatzes.

Examples

Einfache Parametersätze

```
function myFunction
{
    param(
        # If parameter 'a' is used, then 'c' is mandatory
        # If parameter 'b' is used, then 'c' is optional, but allowed
        # You can use parameter 'c' in combination with either 'a' or 'b'
        # 'a' and 'b' cannot be used together

        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [switch]$a,
        [parameter(ParameterSetName="BandC", mandatory=$true)]
        [switch]$b,
        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [parameter(ParameterSetName="BandC", mandatory=$false)]
        [switch]$c
    )
    # $PSCmdlet.ParameterSetName can be used to check which parameter set was used
    Write-Host $PSCmdlet.ParameterSetName
}

# Valid syntaxes
myFunction -a -c
# => "Parameter set : AandC"
myFunction -b -c
# => "Parameter set : BandC"
myFunction -b
# => "Parameter set : BandC"

# Invalid syntaxes
myFunction -a -b
# => "Parameter set cannot be resolved using the specified named parameters."
myFunction -a
# => "Supply values for the following parameters:
#    c:"
```

Parametersatz, um die Verwendung eines Parmeters zu erzwingen, wenn ein anderer ausgewählt wird.

Wenn Sie beispielsweise die Verwendung des Parameters Kennwort erzwingen möchten, wenn der Parameter Benutzer angegeben wird. (und umgekehrt)

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #Do something
}

# This will not work he will ask for user and password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -ComputerName

# This will not work he will ask for password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -User
```

Parametersatz zur Begrenzung der Kombination von Parmetern

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #Do something
}

# This will not work because you can not use the combination Silently and Loudly
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Parametersätze online lesen: https://riptutorial.com/de/powershell/topic/6598/parametersatze

Kapitel 43: PowerShell "Streams"; Debuggen, ausführlich, Warnung, Fehler, Ausgabe und Informationen

Bemerkungen

https://technet.microsoft.com/de-de/library/hh849921.aspx

Examples

Write-Output

`Write-Output` erzeugt eine Ausgabe. Diese Ausgabe kann zum nächsten Befehl nach der Pipeline oder zur Konsole gehen, sodass sie einfach angezeigt wird.

Das Cmdlet sendet Objekte entlang der primären Pipeline, auch "Ausgabestrom" oder "Erfolgspipeline" genannt. Um Fehlerobjekte über die Fehlerpipeline zu senden, verwenden Sie Write-Error.

```
# 1.) Output to the next Cmdlet in the pipeline
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "My name is $_"
}

# 2.) Output to the console since Write-Output is the last command in the pipeline
Write-Output 'Hello world'

# 3.) 'Write-Output' CmdLet missing, but the output is still considered to be 'Write-Output'
'Hello world'
```

1. Das Cmdlet Write-Output sendet das angegebene Objekt in der Pipeline an den nächsten Befehl.
2. Wenn der Befehl der letzte Befehl in der Pipeline ist, wird das Objekt in der Konsole angezeigt.
3. Der PowerShell-Interpreter behandelt dies als implizites Write-Output.

Da bei `Write-Output` standardmäßig die Objekte am Ende einer Pipeline angezeigt werden, ist es normalerweise nicht erforderlich, das Cmdlet zu verwenden. Zum Beispiel `Get-Process | Write-Output` entspricht `Get-Process` .

Schreibe Einstellungen

Mitteilungen können mit geschrieben werden;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Jede davon hat eine Präferenzvariable.

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

Die Einstellungsvariable steuert, wie die Nachricht und die nachfolgende Ausführung des Skripts behandelt werden.

```
$InformationPreference = "SilentlyContinue"
Write-Information "This message will not be shown and execution continues"

$InformationPreference = "Continue"
Write-Information "This message is shown and execution continues"

$InformationPreference = "Inquire"
Write-Information "This message is shown and execution will optionally continue"

$InformationPreference = "Stop"
Write-Information "This message is shown and execution terminates"
```

Die Farbe der Meldungen kann für den `Write-Error` durch Einstellung festgelegt werden.

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Ähnliche Einstellungen sind für `Write-Verbose` , `Write-Debug` und `Write-Warning` verfügbar.

PowerShell "Streams"; Debuggen, ausführlich, Warnung, Fehler, Ausgabe und Informationen online lesen: https://riptutorial.com/de/powershell/topic/3255/powershell--streams---debuggen--ausfuhrlich--warnung--fehler--ausgabe-und-informationen

Kapitel 44: Powershell SQL-Abfragen

Einführung

In diesem Dokument erfahren Sie, wie Sie SQL-Abfragen mit Powershell verwenden

Parameter

Artikel	Beschreibung
$ ServerInstance	Hier muss die Instanz erwähnt werden, in der die Datenbank vorhanden ist
$ Datenbank	Hier müssen wir die Datenbank erwähnen, in der die Tabelle vorhanden ist
$ Query	Hier haben wir die Abfrage, die Sie in SQ ausführen möchten
$ Benutzername & $ Passwort	Benutzername und Passwort, die Zugriff auf die Datenbank haben

Bemerkungen

Sie können die unten stehende Funktion verwenden, wenn Sie kein SQLPS-Modul importieren können

```
function Import-Xls
{

    [CmdletBinding(SupportsShouldProcess=$true)]

    Param(
        [parameter(
            mandatory=$true,
            position=1,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [String[]]
        $Path,

        [parameter(mandatory=$false)]
        $Worksheet = 1,

        [parameter(mandatory=$false)]
        [switch]
        $Force
    )

    Begin
```

```
{
    function GetTempFileName($extension)
    {
        $temp = [io.path]::GetTempFileName();
        $params = @{
            Path = $temp;
            Destination = $temp + $extension;
            Confirm = $false;
            Verbose = $VerbosePreference;
        }
        Move-Item @params;
        $temp += $extension;
        return $temp;
    }

    # since an extension like .xls can have multiple formats, this
    # will need to be changed
    #
    $xlFileFormats = @{
        # single worksheet formats
        '.csv'  = 6;        # 6, 22, 23, 24
        '.dbf'  = 11;       # 7, 8, 11
        '.dif'  = 9;        #
        '.prn'  = 36;       #
        '.slk'  = 2;        # 2, 10
        '.wk1'  = 31;       # 5, 30, 31
        '.wk3'  = 32;       # 15, 32
        '.wk4'  = 38;       #
        '.wks'  = 4;        #
        '.xlw'  = 35;       #

        # multiple worksheet formats
        '.xls'  = -4143;    # -4143, 1, 16, 18, 29, 33, 39, 43
        '.xlsb' = 50;       #
        '.xlsm' = 52;       #
        '.xlsx' = 51;       #
        '.xml'  = 46;       #
        '.ods'  = 60;       #
    }

    $xl = New-Object -ComObject Excel.Application;
    $xl.DisplayAlerts = $false;
    $xl.Visible = $false;
}

Process
{
    $Path | ForEach-Object {

        if ($Force -or $psCmdlet.ShouldProcess($_)) {

            $fileExist = Test-Path $_

            if (-not $fileExist) {
                Write-Error "Error: $_ does not exist" -Category ResourceUnavailable;

            } else {
                # create temporary .csv file from excel file and import .csv
                #
                $_ = (Resolve-Path $_).toString();
                $wb = $xl.Workbooks.Add($_);
```

```
                    if ($?) {
                        $csvTemp = GetTempFileName(".csv");
                        $ws = $wb.Worksheets.Item($Worksheet);
                        $ws.SaveAs($csvTemp, $xlFileFormats[".csv"]);
                        $wb.Close($false);
                        Remove-Variable -Name ('ws', 'wb') -Confirm:$false;
                        Import-Csv $csvTemp;
                        Remove-Item $csvTemp -Confirm:$false -Verbose:$VerbosePreference;
                    }
                }
            }
        }
    }

    End
    {
        $xl.Quit();
        Remove-Variable -name xl -Confirm:$false;
        [gc]::Collect();
    }
}
```

Examples

SQLEexample

Um alle Daten aus der Tabelle *MachineName abzufragen* , können Sie den folgenden Befehl verwenden.

$ Query = "Select * from MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "Benutzer-ID"

$ Password = "Passwort"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

SQLQuery

Um alle Daten aus der Tabelle *MachineName abzufragen* , können Sie den folgenden Befehl verwenden.

$ Query = "Select * from MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "Benutzer-ID"

$ Password = "Passwort"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

Powershell SQL-Abfragen online lesen:
https://riptutorial.com/de/powershell/topic/8217/powershell-sql-abfragen

Kapitel 45: PowerShell.exe Befehlszeile

Parameter

Parameter	Beschreibung
-Hilfe | -? | /?	Zeigt die Hilfe an
-Datei <FilePath> [<Args>]	Pfad zur auszuführenden Skriptdatei und Argumente (optional)
-Befehl {- | <Skriptblock> [-args <Arg-Array>] | <string> [<Befehlsparameter>]}	Auszuführende Befehle gefolgt von Argumenten
-EncodedCommand <Base64EncodedCommand>	Base64-codierte Befehle
-ExecutionPolicy <ExecutionPolicy>	Legt die Ausführungsrichtlinie nur für diesen Prozess fest
-InputFormat {Text | XML}	Legt das Eingabeformat für an den Prozess gesendete Daten fest. Text (Zeichenfolgen) oder XML (serialisierte CLIXML)
-Mta	PowerShell 3.0+: Führt PowerShell in einem Apartment mit mehreren Threads aus (STA ist die Standardeinstellung).
-Sta	PowerShell 2.0: Führt PowerShell in einem Apartment mit einem einzigen Thread aus (MTA ist Standard).
-Kein Ausgang	Lässt die PowerShell-Konsole nach der Ausführung des Skripts / des Befehls ausgeführt
-Kein Logo	Verbirgt das Copyright-Banner beim Start
-NonInteractive	Verbirgt die Konsole vor dem Benutzer
-Kein Profil	Vermeiden Sie das Laden von PowerShell-Profilen für Computer oder Benutzer
-OutputFormat {Text | XML}	Legt das Ausgabeformat für von PowerShell zurückgegebene Daten fest. Text (Zeichenfolgen) oder XML (serialisierte CLIXML)
-PSConsoleFile <Dateipfad>	Lädt eine vorab erstellte Konsolendatei, die die

Parameter	Beschreibung
	Umgebung konfiguriert (erstellt mit `Export-Console`).
-Version <Windows PowerShell-Version>	Geben Sie eine Version von PowerShell zum Ausführen an. Meistens mit `2.0`
-WindowStyle <Stil>	Gibt an, ob der PowerShell-Prozess als `normal` , `hidden` , `minimized` oder `maximized` Fenster gestartet werden `maximized` .

Examples

Befehl ausführen

Mit dem `-Command` Parameter werden Befehle angegeben, die beim Start ausgeführt werden sollen. Es unterstützt mehrere Dateneingaben.

-Command <string>

Sie können Befehle angeben, die beim Start als Zeichenfolge ausgeführt werden sollen. Mehrere Semikolons ; -separierte Anweisungen können ausgeführt werden.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Befehl {Skriptblock}

Der `-Command` Parameter unterstützt auch eine Skriptblock-Eingabe (eine oder mehrere Anweisungen in geschweiften Klammern `{ #code }` . Dies funktioniert nur beim Aufruf von `PowerShell.exe` von einer anderen Windows PowerShell-Sitzung aus.

```
PS > powershell.exe -Command {
"This can be useful, sometimes..."
(Get-Date).ToShortDateString()
}
This can be useful, sometimes...
10.09.2016
```

-Befehl - (Standardeingabe)

Sie können Befehle aus der Standardeingabe mit `-Command -` . Die Standardeingabe kann von `echo` , Lesen einer Datei, einer älteren Konsolenanwendung usw. stammen.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
Greetings from PowerShell
```

Skriptdatei ausführen

Sie können eine Datei an einen angeben `ps1` -script bis es beim Start Inhalt führen die Verwendung `-File` Parameter.

Grundlegendes Skript

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

Ausgabe:

```
>PowerShell.exe -File Desktop\MyScript.ps1
10.09.2016
Hello World
```

Verwenden von Parametern und Argumenten

Sie können Parameter und / oder Argumente nach dem Dateipfad hinzufügen, um sie im Skript zu verwenden. Argumente werden als Werte für nicht definierte / verfügbare Skriptparameter verwendet, der Rest wird im `$args` -array verfügbar sein

MyScript.ps1

```
param($Name)

"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"
"First arg: $($args[0])"
```

Ausgabe:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
Hello StackOverflow! Today's date it 10.09.2016
First arg: foo
```

PowerShell.exe Befehlszeile online lesen:
https://riptutorial.com/de/powershell/topic/5839/powershell-exe-befehlszeile

Kapitel 46: PowerShell-Funktionen

Einführung

Eine Funktion ist im Wesentlichen ein benannter Codeblock. Wenn Sie den Funktionsnamen aufrufen, wird der Skriptblock innerhalb dieser Funktion ausgeführt. Es ist eine Liste von PowerShell-Anweisungen, deren Name Sie zuweisen. Wenn Sie eine Funktion ausführen, geben Sie den Funktionsnamen ein. Mit dieser Methode können Sie Zeit sparen, wenn Sie sich wiederholende Aufgaben lösen. PowerShell formatiert sich in drei Teilen: Das Schlüsselwort 'Function', gefolgt von einem Namen, und schließlich die Nutzdaten, die den Skriptblock enthalten, der von einer geschweiften Klammer mit geschweiften Klammern umgeben ist.

Examples

Einfache Funktion ohne Parameter

Dies ist ein Beispiel für eine Funktion, die eine Zeichenfolge zurückgibt. Im Beispiel wird die Funktion in einer Anweisung aufgerufen, die einer Variablen einen Wert zuweist. Der Wert ist in diesem Fall der Rückgabewert der Funktion.

```
function Get-Greeting{
    "Hello World"
}

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` deklariert den folgenden Code als Funktion.

`Get-Greeting` ist der Name der Funktion. Jedes Mal, wenn diese Funktion im Skript verwendet werden muss, kann die Funktion aufgerufen werden, indem sie nach Namen aufgerufen wird.

`{ ... }` ist der Skriptblock, der von der Funktion ausgeführt wird.

Wenn der obige Code in der ISE ausgeführt wird, würden die Ergebnisse wie folgt aussehen:

```
Hello World
Hello World
```

Grundparameter

Eine Funktion kann mit Parametern über den Parameterblock definiert werden:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory,Position=0)]
        [String]$name,
        [Parameter(Mandatory,Position=1)]
        [Int]$age
    )
    "Hello $name, you are $age years old."
}
```

Oder mit der einfachen Funktionssyntax:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Hinweis: Casting-Parameter sind für beide Arten von Parameterdefinitionen nicht erforderlich.

Die Simple Function Syntax (SFS) hat im Vergleich zum Parameterblock sehr eingeschränkte Möglichkeiten.
Sie können zwar Parameter definieren, die in der Funktion `[CmdletBinding()]` , Sie können jedoch keine Parameterattribute angeben, Parameter Validation verwenden , `[CmdletBinding()]` mit SFS (und dies ist keine erschöpfende Liste) verwenden.

Funktionen können mit geordneten oder benannten Parametern aufgerufen werden.

Die Reihenfolge der Parameter beim Aufruf entspricht der Reihenfolge der Deklaration im Funktionsheader (standardmäßig) oder kann mithilfe des `Position` angegeben werden (wie im Beispiel für erweiterte Funktionen oben gezeigt).

```
$greeting = Write-Greeting "Jim" 82
```

Alternativ kann diese Funktion mit benannten Parametern aufgerufen werden

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Pflichtparameter

Parameter einer Funktion können als obligatorisch markiert werden

```
function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}
```

Wenn die Funktion ohne Wert aufgerufen wird, werden Sie in der Befehlszeile nach dem Wert gefragt:

```
$greeting = Get-Greeting

cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Erweiterte Funktion

Dies ist eine Kopie des erweiterten Funktions-Snippets der Powershell ISE. Im Grunde ist dies eine Vorlage für viele der Dinge, die Sie mit erweiterten Funktionen in Powershell verwenden können. Wichtige Punkte zu beachten:

- get-help-Integration: Der Anfang der Funktion enthält einen Kommentarblock, der so eingerichtet ist, dass er vom Cmdlet get-help gelesen werden kann. Der Funktionsblock kann sich am Ende befinden, falls gewünscht.
- Die Funktion "Cmdletbinding" verhält sich wie ein Cmdlet
- Parameter
- Parametersätze

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                    ValueFromPipeline=$true,
                    ValueFromPipelineByPropertyName=$true,
```

```
                    ValueFromRemainingArguments=$false,
                    Position=0,
                    ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Parameterüberprüfung

Es gibt verschiedene Möglichkeiten, um die Parametereintragung in PowerShell zu überprüfen.

Anstatt Code in Funktionen oder Skripts zu schreiben, um Parameterwerte zu überprüfen, werden diese ParameterAttributes ausgegeben, wenn ungültige Werte übergeben werden.

ValidateSet

Manchmal müssen wir die möglichen Werte einschränken, die ein Parameter annehmen kann. Angenommen, wir möchten für den Parameter `$Color` in einem Skript oder einer Funktion nur Rot, Grün und Blau zulassen.

Wir können das Parameterattribut `ValidateSet` , um dies einzuschränken. Dies hat den

zusätzlichen Vorteil, dass beim Festlegen dieses Arguments (in einigen Umgebungen) das Ausfüllen der Registerkarten zugelassen wird.

```
param(
    [ValidateSet('red','green','blue',IgnoreCase)]
    [string]$Color
)
```

Sie können auch `IgnoreCase` angeben, um die Groß- / Kleinschreibung zu deaktivieren.

ValidateRange

Diese Methode der Parametervalidierung erfordert einen minimalen und maximalen Int32-Wert und erfordert, dass der Parameter innerhalb dieses Bereichs liegt.

```
param(
    [ValidateRange(0,120)]
    [Int]$Age
)
```

ValidatePattern

Diese Methode der Parameterüberprüfung akzeptiert Parameter, die dem angegebenen Regex-Muster entsprechen.

```
param(
    [ValidatePattern("\w{4-6}\d{2}")]
    [string]$UserName
)
```

ValidateLength

Diese Methode der Parameterüberprüfung prüft die Länge der übergebenen Zeichenfolge.

```
param(
    [ValidateLength(0,15)]
    [String]$PhoneNumber
)
```

ValidateCount

Diese Methode der Parameterüberprüfung prüft die Anzahl der übergebenen Argumente, z. B. in einem String-Array.

```
param(
    [ValidateCount(1,5)]
    [String[]]$ComputerName
)
```

ValidateScript

Schließlich ist die ValidateScript-Methode außerordentlich flexibel, indem ein Skriptblock mit $ _ ausgewertet wird, um das übergebene Argument darzustellen. Es übergibt dann das Argument, wenn das Ergebnis $ true ist (einschließlich aller Ausgaben als gültig).

Dies kann verwendet werden, um zu testen, ob eine Datei vorhanden ist:

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

So prüfen Sie, ob ein Benutzer in AD vorhanden ist:

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

Und so ziemlich alles, was Sie schreiben können (da es nicht auf Oneliners beschränkt ist):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

PowerShell-Funktionen online lesen: https://riptutorial.com/de/powershell/topic/1673/powershell-funktionen

Kapitel 47: PowerShell-Hintergrundjobs

Einführung

Jobs wurden in PowerShell 2.0 eingeführt und trugen dazu bei, ein Problem der Befehlszeilentools zu lösen. Kurz gesagt, wenn Sie eine lange laufende Aufgabe starten, ist Ihre Aufforderung erst verfügbar, wenn die Aufgabe abgeschlossen ist. Als Beispiel für eine lange laufende Aufgabe sollten Sie sich diesen einfachen PowerShell-Befehl vorstellen:

Get-ChildItem -Pfad c: \ -Recurse

Es dauert eine Weile, bis die vollständige Verzeichnisliste Ihres Laufwerks C: abgerufen wird. Wenn Sie es als Job ausführen, erhält die Konsole die Kontrolle zurück und Sie können das Ergebnis später erfassen.

Bemerkungen

PowerShell-Jobs werden in einem neuen Prozess ausgeführt. Dies hat Vor- und Nachteile, die miteinander verbunden sind.

Pros:

1. Der Job wird in einem sauberen Prozess ausgeführt, einschließlich Umgebung.
2. Der Job kann asynchron zu Ihrem PowerShell-Hauptprozess ausgeführt werden

Nachteile:

1. Änderungen in der Prozessumgebung sind im Job nicht vorhanden.
2. Parameterübergabe an und zurückgegebene Ergebnisse werden serialisiert.
    - Das heißt, wenn Sie ein Parameterobjekt ändern, während der Job ausgeführt wird, wird es nicht im Job angezeigt.
    - Dies bedeutet auch, wenn ein Objekt nicht serialisiert werden kann, können Sie es nicht übergeben oder zurückgeben (obwohl PowerShell Parameter kopieren und ein PSObject übergeben / übergeben kann.)

Examples

Grundlegende Schaffung von Arbeitsplätzen

Starten Sie einen Scriptblock als Hintergrundjob:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Starten Sie ein Skript als Hintergrundjob:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Starten Sie einen Job mit `Invoke-Command` auf einem Remote-Computer:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName
"WinRM" -ThrottleLimit 16 -AsJob
```

Job als anderer Benutzer starten (Aufforderung zur Eingabe eines Passworts):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

Oder

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Job als anderer Benutzer starten (Keine Eingabeaufforderung):

```
$username = "Domain\Username"
$password = "password"
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,
$secPassword)
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Grundlegende Jobverwaltung

Rufen Sie eine Liste aller Jobs in der aktuellen Sitzung ab:

```
Get-Job
```

Warten auf einen Job, um den Vorgang abzuschließen, bevor Sie das Ergebnis erhalten:

```
$job | Wait-job | Receive-Job
```

Timeout eines Jobs, wenn er zu lang ist (in diesem Beispiel 10 Sekunden)

```
$job | Wait-job -Timeout 10
```

Job anhalten (erledigt alle Aufgaben, die vor dem Beenden in der Jobwarteschlange anstehen)

```
$job | Stop-Job
```

Job aus der Hintergrund-Jobliste der aktuellen Sitzung entfernen:

```
$job | Remove-Job
```

Hinweis : Folgendes funktioniert nur bei `Workflow` Jobs.

Einen `Workflow` Job `Workflow` (Pause):

```
$job | Suspend-Job
```

Einen `Workflow` Job wieder aufnehmen:

```
$job | Resume-Job
```

PowerShell-Hintergrundjobs online lesen:
https://riptutorial.com/de/powershell/topic/3970/powershell-hintergrundjobs

Kapitel 48: PowerShell-Klassen

Einführung

Eine Klasse ist eine erweiterbare Programmcode-Vorlage zum Erstellen von Objekten, zur Bereitstellung von Anfangswerten für Status (Elementvariablen) und Implementierungen von Verhalten (Elementfunktionen oder -methoden). Eine Klasse ist ein Entwurf für ein Objekt. Es wird als Modell verwendet, um die Struktur von Objekten zu definieren. Ein Objekt enthält Daten, auf die wir über Eigenschaften zugreifen und an denen mit Methoden gearbeitet werden kann. In PowerShell 5.0 wurde die Möglichkeit hinzugefügt, eigene Klassen zu erstellen.

Examples

Methoden und Eigenschaften

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Verfügbare Konstruktoren für eine Klasse auflisten

5,0

In PowerShell 5.0+ können Sie verfügbare Konstruktoren auflisten, indem Sie die statische `new` Methode ohne Klammern aufrufen.

```
PS> [DateTime]::new

OverloadDefinitions
-------------------
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
```

```
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

Mit dieser Methode können Sie Überladungsdefinitionen für jede Methode auflisten

```
> 'abc'.CompareTo

OverloadDefinitions
-------------------
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)
```

Für frühere Versionen können Sie eine eigene Funktion erstellen, um verfügbare Konstruktoren aufzulisten:

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
            e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}
```

Verwendungszweck:

```
Get-Constructor System.DateTime
#Or [datetime] | Get-Constructor

DateTime Constructors
---------------------
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
```

```
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Konstruktor überladen

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

Holen Sie sich alle Mitglieder einer Instanz

```
PS > Get-Member -InputObject $anObjectInstance
```

Dadurch werden alle Mitglieder der Typinstanz zurückgegeben. Hier ist ein Teil einer Beispielausgabe für die String-Instanz

```
   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method                int CompareTo(System.Object value), int
CompareTo(string strB), i...
Contains         Method                bool Contains(string value)
CopyTo           Method                void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith         Method                bool EndsWith(string value), bool EndsWith(string
value, System.S...
Equals           Method                bool Equals(System.Object obj), bool Equals(string
value), bool E...
GetEnumerator    Method                System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode      Method                int GetHashCode()
GetType          Method                type GetType()
...
```

Grundlegende Klassenvorlage

```
# Define a class
class TypeName
{
   # Property with validate set
```

```
    [ValidateSet("val1", "Val2")]
    [string] $P1

    # Static property
    static [hashtable] $P2

    # Hidden property does not show as result of Get-Member
    hidden [int] $P3

    # Constructor
    TypeName ([string] $s)
    {
        $this.P1 = $s
    }

    # Static method
    static [void] MemberMethod1([hashtable] $h)
    {
        [TypeName]::P2 = $h
    }

    # Instance method
    [int] MemberMethod2([int] $i)
    {
        $this.P3 = $i
        return $this.P3
    }
}
```

Vererbung von übergeordneter Klasse zu untergeordneter Klasse

```
class ParentClass
{
    [string] $Message = "Its under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}

# Bar extends Foo and inherits its members
class ChildClass : ParentClass
{

}
$Inherit = [ChildClass]::new()
```

Also , **$ Inherit.Message** gibt Ihnen die

> "Es ist unter der Elternklasse"

PowerShell-Klassen online lesen: https://riptutorial.com/de/powershell/topic/1146/powershell-klassen

Kapitel 49: Powershell-Module

Einführung

Mit der PowerShell-Version 2.0 können Entwickler PowerShell-Module erstellen. PowerShell-Module kapseln eine Reihe allgemeiner Funktionen. Beispielsweise gibt es herstellerspezifische PowerShell-Module, die verschiedene Cloud-Services verwalten. Es gibt auch allgemeine PowerShell-Module, die mit Social-Media-Diensten interagieren und gängige Programmieraufgaben ausführen, z. B. Base64-Codierung, Arbeiten mit Named Pipes usw.

Module können Befehlsaliase, Funktionen, Variablen, Klassen und mehr verfügbar machen.

Examples

Erstellen Sie ein Modulmanifest

```
@{
  RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
  CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
  CompanyName = 'My Company'
  Copyright = '(c) 2017 Administrator. All rights reserved.'
  Description = 'It does cool stuff.'
  FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

Jedes gute PowerShell-Modul verfügt über ein Modulmanifest. Das Modulmanifest enthält lediglich Metadaten zu einem PowerShell-Modul und definiert nicht den tatsächlichen Inhalt des Moduls.

Die Manifestdatei ist eine PowerShell-Skriptdatei mit der Erweiterung `.psd1` , die eine HashTable enthält. Die HashTable im Manifest muss bestimmte Schlüssel enthalten, damit PowerShell sie als PowerShell-Moduldatei richtig interpretieren kann.

Das obige Beispiel enthält eine Liste der wichtigsten HashTable-Schlüssel, aus denen ein Modulmanifest besteht, es gibt jedoch noch viele andere. Mit dem Befehl `New-ModuleManifest` Sie ein neues Skelett für das `New-ModuleManifest` erstellen.

Einfaches Modulbeispiel

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
```

```
  , [int] $y
  )

  return $x + $y
}

Export-ModuleMember -Function Add
```

Dies ist ein einfaches Beispiel, wie eine PowerShell-Skriptmodul-Datei aussehen könnte. Diese Datei würde `MyCoolModule.psm1` heißen und wird aus der Modulmanifestdatei (.psd1) referenziert. Sie werden feststellen, dass wir mit dem Befehl `Export-ModuleMember` angeben können, welche Funktionen in dem Modul für den Benutzer des Moduls "exportiert" werden sollen. Einige Funktionen sind nur für den internen `Export-ModuleMember` und sollten nicht `Export-ModuleMember` werden, sodass sie beim Aufruf von `Export-ModuleMember` .

Eine Variable aus einem Modul exportieren

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

Um eine Variable aus einem Modul zu exportieren, verwenden Sie den Befehl `Export-ModuleMember` mit dem Parameter `-Variable` . Denken Sie jedoch daran, dass, wenn die Variable auch nicht explizit in der Modulmanifestdatei (.psd1) exportiert wird, die Variable für den Modulverbraucher nicht sichtbar ist. Stellen Sie sich das Manifest des Moduls wie einen "Gatekeeper" vor. Wenn eine Funktion oder Variable im Modulmanifest nicht zulässig ist, ist sie für den Modulverbraucher nicht sichtbar.

Anmerkung: Das Exportieren einer Variablen ähnelt dem Veröffentlichen eines Felds in einer Klasse. Es ist nicht ratsam. Es wäre besser, eine Funktion freizulegen, um das Feld zu erhalten, und eine Funktion, um das Feld festzulegen.

Strukturieren von PowerShell-Modulen

Anstatt alle Funktionen in einer einzigen PowerShell-Skriptmoduldatei `.psm1` , möchten Sie möglicherweise Ihre Funktion in einzelne Dateien `.psm1` . Sie können diese Dateien dann aus Ihrer Skriptmodul-Datei mit einem Punkt versehen, die sie im Wesentlichen so behandelt, als wären sie Teil der `.psm1` Datei.

Betrachten Sie diese Modulverzeichnisstruktur:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

In Ihre `MyCoolModule.psm1` Datei können Sie folgenden Code einfügen:

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

Dadurch würden die einzelnen Funktionsdateien in die `.psm1` Moduldatei aufgenommen.

Lage der Module

PowerShell sucht nach Modulen in den Verzeichnissen, die unter $ Env: PSModulepath aufgeführt sind.

Ein Modul namens *foo* in einem Ordner namens *foo* wird mit dem `Import-Module foo`

In diesem Ordner sucht PowerShell nach einem Modulmanifest (foo.psd1), einer Moduldatei (foo.psm1) und einer DLL (foo.dll).

Sichtbarkeit von Modulmitgliedern

Standardmäßig sind nur die in einem Modul definierten Funktionen außerhalb des Moduls sichtbar. Wenn Sie also Variablen und Aliase in einem Modul definieren, stehen diese nur im Code des Moduls zur Verfügung.

Um dieses Verhalten zu überschreiben, können Sie das `Export-ModuleMember` verwenden. Mit den Parametern `-Function` , `-Variable` und `-Alias` können Sie genau angeben, welche `-Alias` exportiert werden sollen.

Beachten Sie, dass bei Verwendung von `Export-ModuleMember` **nur** die von Ihnen angegebenen Elemente sichtbar sind.

Powershell-Module online lesen: https://riptutorial.com/de/powershell/topic/8734/powershell-module

Kapitel 50: Powershell-Profile

Bemerkungen

Die Profildatei ist ein Powershell-Skript, das ausgeführt wird, während die Powershell-Konsole gestartet wird. Auf diese Weise können wir jedes Mal, wenn wir eine neue Powershell-Sitzung beginnen, unsere Umgebung für uns vorbereiten.

Typische Dinge, die wir am Powershell-Start machen wollen, sind:

- Importieren von häufig verwendeten Modulen (ActiveDirectory, Exchange, einige spezielle DLL)
- Protokollierung
- Eingabeaufforderung ändern
- Diagnose

Es gibt verschiedene Profildateien und -speicherorte, die unterschiedliche Verwendungszwecke und auch eine Hierarchie der Startreihenfolge haben:

Wirt	Nutzer	Pfad	Bestellung starten	Variable
Alles	Alles	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ profile.ps1	1	$ profile.AllUsersAllHosts
Alles	Aktuell	% USERPROFILE% \ Documents \ WindowsPowerShell \ profile.ps1	3	$ profile.CurrentUserAllHosts
Konsole	Alles	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShell_profile.ps1	2	$ profile.AllUsersCurrentHost
Konsole	Aktuell	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShell_profile.ps1	4	$ profile.CurrentUserCurrentHost
ISE	Alles	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShellISE_profile.ps1	2	$ profile.AllUsersCurrentHost
ISE	Aktuell	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShellISE_profile.ps1	4	$ profile.CurrentUserCurrentHost

Examples

Erstellen Sie ein Basisprofil

Ein PowerShell-Profil wird verwendet, um benutzerdefinierte Variablen und Funktionen automatisch zu laden.

PowerShell-Profile werden nicht automatisch für Benutzer erstellt.

So erstellen Sie ein PowerShell-Profil `C:>New-Item -ItemType File $profile .`

Wenn Sie sich in ISE befinden, können Sie das eingebaute `C:>psEdit $profile`

Ein einfacher Weg, um mit Ihrem persönlichen Profil für den aktuellen Host zu beginnen, besteht darin, Text in einem Pfad zu speichern, der in der `$profile` -Variable gespeichert ist

```
"#Current host, current user" > $profile
```

Weitere Änderungen am Profil können mit PowerShell ISE, Notepad, Visual Studio Code oder einem anderen Editor vorgenommen werden.

Die `$profile` -variable gibt standardmäßig das aktuelle Benutzerprofil für den aktuellen Host zurück. Sie können jedoch auf den Pfad zur Computerrichtlinie (alle Benutzer) und / oder auf das Profil für alle Hosts (Konsole, ISE, Drittanbieter) zugreifen es sind Eigenschaften.

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Powershell-Profile online lesen: https://riptutorial.com/de/powershell/topic/5636/powershell-profile

Kapitel 51: Powershell-Remoting

Bemerkungen

- about_Remote
- about_RemoteFAQ
- about_RemoteFehlerbehebung

Examples

Aktivieren von PowerShell-Remoting

PowerShell-Remoting muss zuerst auf dem Server aktiviert werden, zu dem Sie eine Remote-Verbindung herstellen möchten.

```
Enable-PSRemoting -Force
```

Dieser Befehl führt Folgendes aus:

- Führt das Cmdlet Set-WSManQuickConfig aus, das die folgenden Aufgaben ausführt:
- Startet den WinRM-Dienst.
- Legt den Starttyp für den WinRM-Dienst auf Automatisch fest.
- Erstellt einen Listener, um Anforderungen an eine beliebige IP-Adresse anzunehmen, falls noch keine vorhanden ist.
- Aktiviert eine Firewall-Ausnahme für die WS-Management-Kommunikation.
- Registriert die Microsoft.PowerShell- und Microsoft.PowerShell.Workflow-Sitzungskonfigurationen, sofern diese noch nicht registriert sind.
- Registriert die Microsoft.PowerShell32-Sitzungskonfiguration auf 64-Bit-Computern, sofern diese nicht bereits registriert ist.
- Aktiviert alle Sitzungskonfigurationen.
- Ändert die Sicherheitsbeschreibung aller Sitzungskonfigurationen, um den Fernzugriff zu ermöglichen.
- Startet den WinRM-Dienst neu, damit die vorhergehenden Änderungen wirksam werden.

Nur für Umgebungen außerhalb von Domänen

Bei Servern in einer AD-Domäne erfolgt die PS-Remoting-Authentifizierung über Kerberos ('Default') oder NTLM ('Negotiate'). Wenn Sie Remoting auf einem Nicht-Domänenserver zulassen möchten, haben Sie zwei Möglichkeiten.

Richten Sie entweder die WSMan-Kommunikation über HTTPS ein (dies erfordert die Erzeugung

eines Zertifikats) oder aktivieren Sie die Basisauthentifizierung, die Ihre Anmeldeinformationen über die Basis64-Verschlüsselung sendet.

In beiden Fällen müssen Sie die fernen Systeme der Liste der vertrauenswürdigen WSMan-Hosts hinzufügen.

Aktivieren der Standardauthentifizierung

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Dann auf dem Computer , *von* dem Sie verbinden möchten, müssen Sie es sagen , um den Computer vertrauen Sie eine Verbindung herstellen.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Wichtig : Sie müssen Ihrem Client mitteilen, dass er dem angesprochenen Computer auf die Art und Weise, wie Sie eine Verbindung herstellen möchten, vertrauen soll (z. B. wenn Sie sich über IP verbinden, muss er der IP und nicht dem Hostnamen vertrauen)

Verbindung zu einem Remote-Server über PowerShell

Verwenden von Anmeldeinformationen von Ihrem lokalen Computer:

```
Enter-PSSession 192.168.1.1
```

Aufforderung zur Eingabe von Anmeldeinformationen auf dem Remote-Computer

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Befehle auf einem Remote-Computer ausführen

Sobald Powershell-Remoting aktiviert ist (Enable-PSRemoting), können Sie Befehle auf dem Remote-Computer wie folgt ausführen:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}
```

Die obige Methode erstellt eine temporäre Sitzung und schließt diese unmittelbar nach Beendigung des Befehls oder Skriptblocks.

Um die Sitzung geöffnet zu lassen und später einen anderen Befehl auszuführen, müssen Sie

zuerst eine Remote-Sitzung erstellen:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Sie können diese Sitzung dann jedes Mal verwenden, wenn Sie Befehle auf dem Remote-Computer aufrufen:

```
Invoke-Command -Session $Session -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

Wenn Sie verschiedene Anmeldeinformationen verwenden müssen, können Sie diese mit dem Parameter `-Credential` hinzufügen:

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Serialisierungswarnung für Remoting

> Hinweis:
>
> Es ist wichtig zu wissen, dass beim Remoting PowerShell-Objekte auf dem Remote-System serialisiert und am Ende der Remoting-Sitzung deserialisiert werden, dh sie werden während des Transports in XML konvertiert und verlieren alle ihre Methoden.

```
$output = Invoke-Command -Session $Session -ScriptBlock {
    Get-WmiObject -Class win32_printer
}

$output | Get-Member -MemberType Method

  TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer

Name     MemberType Definition
----     ---------- ----------
GetType  Method     type GetType()
ToString Method     string ToString(), string ToString(string format, System.IFormatProvi...
```

Während Sie die Methoden für das reguläre PS-Objekt haben:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method

 TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name                   MemberType Definition

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

```
CancelAllJobs          Method     System.Management.ManagementBaseObject CancelAllJobs()

GetSecurityDescriptor Method     System.Management.ManagementBaseObject
GetSecurityDescriptor()
Pause                  Method     System.Management.ManagementBaseObject Pause()

PrintTestPage          Method     System.Management.ManagementBaseObject PrintTestPage()

RenamePrinter          Method     System.Management.ManagementBaseObject
RenamePrinter(System.String NewPrinterName)
Reset                  Method     System.Management.ManagementBaseObject Reset()

Resume                 Method     System.Management.ManagementBaseObject Resume()

SetDefaultPrinter      Method     System.Management.ManagementBaseObject SetDefaultPrinter()

SetPowerState          Method     System.Management.ManagementBaseObject
SetPowerState(System.UInt16 PowerState, System.String Time)
SetSecurityDescriptor Method     System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject#Win32_SecurityDescriptor Descriptor)
```

Argumentverwendung

Um Argumente als Parameter für den Remote-Skriptblock zu verwenden, kann man entweder den `ArgumentList` Parameter von `Invoke-Command` oder die `$Using:` -Syntax verwenden.

`ArgumentList` mit unbenannten Parametern verwenden (dh in der Reihenfolge, in der sie an den Scriptblock übergeben werden):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

`ArgumentList` mit benannten Parametern verwenden:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)

    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Verwendung von `$Using:` Syntax:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
```

```
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Eine bewährte Methode zum automatischen Aufräumen von PSSessions

Wenn eine Remote-Sitzung über das `New-PSsession` erstellt wird, bleibt die PSSession bestehen, bis die aktuelle PowerShell-Sitzung endet. Das bedeutet, dass standardmäßig die `PSSession` und alle zugeordneten Ressourcen bis zum Ende der aktuellen PowerShell-Sitzung verwendet werden.

Mehrere aktive `PSSessions` können Ressourcen `PSSessions` , insbesondere für langlaufende oder verknüpfte Skripts, die Hunderte von `PSSessions` in einer einzigen PowerShell-Sitzung erstellen.

Es wird `PSSession` , jede `PSSession` nach der Verwendung explizit zu entfernen. [1]

Die folgende Codevorlage verwendet `try-catch-finally` , um die oben genannten `PSSessions` zu erreichen. Sie kombiniert die Fehlerbehandlung mit einer sicheren Methode, um sicherzustellen, dass alle erstellten `PSSessions` entfernt werden, wenn sie fertig verwendet werden:

```
try
{
    $session = New-PSsession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

Verweise: [1] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession

Powershell-Remoting online lesen: https://riptutorial.com/de/powershell/topic/3087/powershell-remoting

Kapitel 52: PowerShell-Workflows

Einführung

PowerShell Workflow ist eine Funktion, die ab PowerShell Version 3.0 eingeführt wurde. Workflow-Definitionen sehen PowerShell-Funktionsdefinitionen sehr ähnlich, werden jedoch innerhalb der Windows Workflow Foundation-Umgebung und nicht direkt in der PowerShell-Engine ausgeführt.

Die Workflow-Engine enthält mehrere einzigartige "Out-of-Box" -Funktionen, vor allem die Persistenz von Jobs.

Bemerkungen

Die PowerShell Workflow-Funktion wird ausschließlich auf der Microsoft Windows-Plattform unter PowerShell Desktop Edition unterstützt. PowerShell Core Edition, die unter Linux, Mac und Windows unterstützt wird, unterstützt die PowerShell Workflow-Funktion nicht.

Beachten Sie beim Erstellen eines PowerShell-Workflows, dass bei Workflows Aktivitäten und nicht Cmdlets aufgerufen werden. Sie können zwar weiterhin Cmdlets von einem PowerShell-Workflow aus aufrufen, die Workflow Engine wird jedoch den Aufruf des Cmdlets implizit in eine `InlineScript` Aktivität `InlineScript` . Sie können Code auch explizit in die `InlineScript` Aktivität `InlineScript` , die PowerShell-Code ausführt. Die `InlineScript` Aktivität wird standardmäßig in einem separaten Prozess ausgeführt und gibt das Ergebnis an den aufrufenden Workflow zurück.

Examples

Einfaches Workflow-Beispiel

```
workflow DoSomeWork {
  Get-Process -Name notepad | Stop-Process
}
```

Dies ist ein grundlegendes Beispiel für eine PowerShell Workflow-Definition.

Workflow mit Eingabeparametern

Genau wie die PowerShell-Funktionen können Workflows Eingabeparameter akzeptieren. Eingabeparameter können optional an einen bestimmten Datentyp gebunden werden, z. B. eine Zeichenfolge, eine Ganzzahl usw. Verwenden Sie das Standardschlüsselwort `param` , um einen Block von Eingabeparametern direkt nach der Workflow-Deklaration zu definieren.

```
workflow DoSomeWork {
  param (
    [string[]] $ComputerName
```

```
  )
  Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

Führen Sie den Workflow als Hintergrundjob aus

PowerShell-Workflows sind grundsätzlich mit der Möglichkeit ausgestattet, als Hintergrundjob ausgeführt zu werden. Verwenden `-AsJob` zum Aufrufen eines Workflows als PowerShell-Hintergrundjob den Parameter `-AsJob` , wenn Sie den Workflow aufrufen.

```
workflow DoSomeWork {
  Get-Process -ComputerName server01
  Get-Process -ComputerName server02
  Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

Fügen Sie einem Workflow einen Parallelblock hinzu

```
workflow DoSomeWork {
  parallel {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
  }
}
```

Eine der einzigartigen Funktionen von PowerShell Workflow ist die Möglichkeit, einen Aktivitätsblock als parallel zu definieren. Um diese Funktion zu verwenden, verwenden Sie das `parallel` Schlüsselwort in Ihrem Workflow.

Durch das parallele Aufrufen von Workflowaktivitäten kann die Leistung Ihres Workflows verbessert werden.

PowerShell-Workflows online lesen: https://riptutorial.com/de/powershell/topic/8745/powershell-workflows

Kapitel 53: PSScriptAnalyzer - PowerShell Script Analyzer

Einführung

PSScriptAnalyzer, https://github.com/PowerShell/PSScriptAnalyzer , ist ein statischer Code-Checker für Windows PowerShell-Module und -Scripts. PSScriptAnalyzer überprüft die Qualität von Windows PowerShell-Code, indem es eine Reihe von Regeln ausführt, die auf den vom PowerShell-Team und der Community ermittelten Best Practices basieren. Es generiert Diagnose-Ergebnisse (Fehler und Warnungen), um Benutzer über mögliche Codefehler zu informieren, und schlägt mögliche Lösungen für Verbesserungen vor.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Syntax

1. `Get-ScriptAnalyzerRule [-CustomizedRulePath <string[]>] [-Name <string[]>] [-Severity <string[]>] [<CommonParameters>]`
2. `Invoke-ScriptAnalyzer [-Path] <string> [-CustomizedRulePath <string[]>] [-ExcludeRule <string[]>] [-IncludeRule<string[]>] [-Severity <string[]>] [-Recurse] [-SuppressedOnly] [<CommonParameters>]`

Examples

Analysieren von Skripts mit den integrierten voreingestellten Regelsätzen

Im Lieferumfang von ScriptAnalyzer sind vordefinierte Regeln enthalten, die zur Analyse von Skripts verwendet werden können. Dazu gehören: `PSGallery` , `DSC` und `CodeFormatting` . Sie können wie folgt ausgeführt werden:

Regeln für die PowerShell-Galerie

Verwenden Sie zum Ausführen der PowerShell Gallery-Regeln den folgenden Befehl:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

DSC-Regeln

Um die DSC-Regeln auszuführen, verwenden Sie den folgenden Befehl:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Code-Formatierungsregeln

Verwenden Sie zum Ausführen der Code-Formatierungsregeln den folgenden Befehl:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Analysieren von Skripts anhand jeder integrierten Regel

So führen Sie den Skriptanalysator für eine einzelne Skriptdatei aus:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Dadurch wird Ihr Skript anhand jeder integrierten Regel analysiert. Wenn Ihr Skript ausreichend groß ist, kann dies zu zahlreichen Warnungen und / oder Fehlern führen.

Um den Skriptanalysator für ein gesamtes Verzeichnis auszuführen, geben Sie den Ordner an, der die zu analysierenden Skript-, Modul- und DSC-Dateien enthält. Geben Sie den Parameter Recurse an, wenn Sie auch die Unterverzeichnisse nach zu analysierenden Dateien durchsuchen möchten.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Listet alle integrierten Regeln auf

Um zu sehen, wie alle integrierten Regeln ausgeführt werden:

```
Get-ScriptAnalyzerRule
```

PSScriptAnalyzer - PowerShell Script Analyzer online lesen:
https://riptutorial.com/de/powershell/topic/9619/psscriptanalyzer---powershell-script-analyzer

Kapitel 54: Regeln der Namensgebung

Examples

Funktionen

```
Get-User()
```

- Verwenden Sie das *Verb-Noun*- Muster, wenn Sie eine Funktion benennen.
- Verb impliziert eine Aktion, zB `Get` , `Set` , `New` , `Read` , `Write` und viele mehr. Siehe genehmigte Verben .
- Das Substantiv sollte singulär sein, auch wenn es auf mehrere Elemente wirkt. `Get-User()` kann einen oder mehrere Benutzer zurückgeben.
- Verwenden Sie den Pascal-Fall für Verb und Noun. ZB `Get-UserLogin()`

Regeln der Namensgebung online lesen: https://riptutorial.com/de/powershell/topic/9714/regeln-der-namensgebung

Kapitel 55: Reguläre Ausdrücke

Syntax

- 'Text' - Übereinstimmung mit 'RegExPattern'
- 'text' -replace 'RegExPattern', 'newvalue'
- [Regex] :: Match ("Text", "Muster") #Einzeln
- [Regex] :: Übereinstimmungen ("Text", "Muster") #Multiple Übereinstimmungen
- [Regex] :: Ersetzen ("Text", "Muster", "Neuer Wert")
- [Regex] :: Ersetzen ("Text", "Muster", {Param ($ m)}) #MatchEvaluator
- [Regex] :: Escape ("Eingabe") #Escape-Sonderzeichen

Examples

Einziges Paar

Mit Regex können Sie schnell feststellen, ob ein Text ein bestimmtes Muster enthält. Es gibt mehrere Möglichkeiten, mit Regex in PowerShell zu arbeiten.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Verwenden des -Match-Operators

Verwenden Sie die Syntax `'input' -match 'pattern'` , um festzustellen, ob eine Zeichenfolge mit dem integrierten `-matches` Operator `'input' -match 'pattern'` . Dies gibt je nach Ergebnis der Suche " `true` oder " `false` . Wenn Übereinstimmung vorhanden ist, können Sie die Übereinstimmung und die Gruppen (sofern in Muster definiert) anzeigen, indem Sie auf die Variable `$Matches` zugreifen.

```
> $text -match $pattern
True

> $Matches

Name Value
---- -----
0    (a)
```

Sie können auch `-match` , um durch ein Array von Strings zu filtern und nur die Strings zurückzugeben, die eine Übereinstimmung enthalten.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"

> $textarray -match $pattern
This is (a) sample
a (sample text)
```

2,0

Select-String verwenden

In PowerShell 2.0 wurde ein neues Cmdlet für das Durchsuchen von Text mit Regex eingeführt. Es gibt ein `MatchInfo` Objekt pro Texteingabe zurück, die eine Übereinstimmung enthält. Sie können auf seine Eigenschaften zugreifen, um übereinstimmende Gruppen usw. zu finden.

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

This is (a) sample
text, this is
a (sample text)

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}
```

Genau wie `-match` kann `Select-String` auch verwendet werden, um ein Array von Strings zu filtern, indem ein Array an dieses übergeben wird. Es erstellt ein `MatchInfo` -Objekt pro Zeichenfolge, das eine Übereinstimmung enthält.

```
> $textarray | Select-String -Pattern $pattern

This is (a) sample
a (sample text)

#You can also access the matches, groups etc.
> $textarray | Select-String -Pattern $pattern | fl *
```

```
IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}

IgnoreCase : True
LineNumber : 3
Line       : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(sample text)}
```

`Select-String` kann auch mit einem normalen Textmuster (ohne `-SimpleMatch`) `-SimpleMatch` , indem Sie den Schalter `-SimpleMatch` hinzufügen.

Verwendung von [RegEx] :: Match ()

Sie können auch die statische `Match()` -Methode verwenden, die in der .NET `[RegEx]` -Klasse verfügbar ist.

```
> [regex]::Match($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

Ersetzen

Eine übliche Aufgabe für Regex ist das Ersetzen von Text, der einem Muster entspricht, durch einen neuen Wert.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Text wrapped in ()
$pattern = '\(.*?\)'
```

```
#Replace matches with:
$newvalue = 'test'
```

Verwenden Sie -Replace Operator

Der `-replace` Operator in PowerShell kann verwendet werden, um Text, der einem Muster entspricht, durch einen neuen Wert zu ersetzen. Verwenden Sie dazu die Syntax `'input' -replace 'pattern', 'newvalue'` .

```
> $text -replace $pattern, $newvalue
This is test sample
text, this is
a test
```

Verwendung der Methode [RegEx] :: Replace ()

Das Ersetzen von Übereinstimmungen kann auch mit der `Replace()` Methode in der .NET-Klasse `[RegEx]` .

```
[regex]::Replace($text, $pattern, 'test')
This is test sample
text, this is
a test
```

Ersetzen Sie Text durch einen dynamischen Wert mit einem MatchEvalutor

In manchen Fällen müssen Sie einen Wert, der einem Muster entspricht, durch einen neuen Wert ersetzen, der auf diesem bestimmten Treffer basiert, sodass der neue Wert nicht vorhergesagt werden kann. Für diese Arten von Szenarien kann ein `MatchEvaluator` sehr nützlich sein.

In PowerShell ist ein `MatchEvaluator` so einfach wie ein Skriptblock mit einem einzigen Parameter, der ein `Match` Objekt für das aktuelle Match enthält. Die Ausgabe der Aktion ist der neue Wert für diese bestimmte Übereinstimmung. `MatchEvalutor` kann mit der statischen Methode `[Regex]::Replace()` verwendet werden.

Beispiel : Ersetzen des Textes inside `()` durch seine Länge

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '(?<=\().*?(?=\))'
```

```
$MatchEvalutor = {
    param($match)

    #Replace content with length of content
    $match.Value.Length

}
```

Ausgabe:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is 1 sample
text, this is
a 11
```

Beispiel: Machen Sie ein `sample` Großbuchstaben

```
#Sample pattern: "Sample"
$pattern = 'sample'

$MatchEvalutor = {
    param($match)

    #Return match in upper-case
    $match.Value.ToUpper()

}
```

Ausgabe:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Sonderzeichen entkommen

Ein Regex-Muster verwendet viele Sonderzeichen, um ein Muster zu beschreiben. Ex., `.` bedeutet "beliebiges Zeichen", `+` ist "ein oder mehrere" usw.

Um diese Zeichen zu verwenden, als `.` , `+` usw. in einem Muster müssen Sie sie entziehen, um ihre spezielle Bedeutung zu entfernen. Dazu verwenden Sie das Escape-Zeichen, das in regex ein Backslash `\` . Beispiel: Um nach `+` zu suchen, verwenden Sie das Muster `\+` .

Es kann schwierig sein, sich alle Sonderzeichen in Regex zu merken. Um jedes Sonderzeichen in einer Zeichenfolge zu speichern, nach der Sie suchen möchten, können Sie die Methode `[RegEx]::Escape("input")` verwenden.

```
> [regex]::Escape("(foo)")
\(foo\)
```

```
> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Mehrere Übereinstimmungen

Es gibt mehrere Möglichkeiten, alle Übereinstimmungen für ein Muster in einem Text zu finden.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Select-String verwenden

Sie können alle Übereinstimmungen (globale Übereinstimmung) finden, indem Sie den Schalter `-AllMatches` zu `Select-String` hinzufügen.

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a), (sample text)}

#List all matches
> $m.Matches

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)
```

```
#Get matched text
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Verwendung von [RegEx] :: Matches ()

Die `Matches()` Methode in der .NET-Klasse [regex] kann auch verwendet werden, um eine globale Suche nach mehreren Übereinstimmungen durchzuführen.

```
> [regex]::Matches($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

Reguläre Ausdrücke online lesen: https://riptutorial.com/de/powershell/topic/6674/regulare-ausdrucke

Kapitel 56: Schleifen

Einführung

Eine Schleife ist eine Befehlsfolge, die ständig wiederholt wird, bis eine bestimmte Bedingung erreicht ist. In der Lage zu sein, Ihr Programm wiederholt einen Codeblock ausführen zu lassen, ist eine der grundlegendsten, aber nützlichsten Aufgaben bei der Programmierung. In einer Schleife können Sie eine sehr einfache Anweisung schreiben, um durch Wiederholung ein deutlich größeres Ergebnis zu erzielen. Wenn die Bedingung erreicht ist, "fällt" der nächste Befehl zum nächsten sequentiellen Befehl oder verzweigt außerhalb der Schleife.

Syntax

- für (<Initialisierung>; <Bedingung>; <Wiederholung>) {<Script_Block>}
- <Sammlung> | Foreach-Objekt {<Script_Block_with _ $ __ as_current_item>}
- foreach (<item> in <Collection>) {<Skriptblock>}
- while (<Bedingung>) {<Script_Block>}
- do {<Script_Block>} while (<Bedingung>)
- do {<Script_Block>} bis (<Bedingung>)
- <Collection> .foreach ({<Script_Block_with _ $ __ as_current_item>})

Bemerkungen

Für jeden

Es gibt mehrere Möglichkeiten, eine Foreach-Schleife in PowerShell auszuführen, und alle bringen ihre eigenen Vor- und Nachteile mit sich:

Lösung	Vorteile	Nachteile
Foreach-Anweisung	Am schnellsten Funktioniert am besten mit statischen Sammlungen (in einer Variablen gespeichert).	Keine Pipeline-Eingabe oder -Ausgabe
ForEach () -Methode	Dieselbe Skriptblock-Syntax wie `Foreach-Object` , jedoch schneller. Funktioniert am besten mit statischen Sammlungen (in einer Variablen gespeichert). Unterstützt die Pipeline-Ausgabe.	Keine Unterstützung für Pipeline-Eingabe. Erfordert PowerShell 4.0 oder höher

Lösung	Vorteile	Nachteile
Foreach-Object (Cmdlet)	Unterstützt die Eingabe und Ausgabe von Pipelines. Unterstützt Anfangs- und End-Skriptblöcke zum Initialisieren und Schließen von Verbindungen usw. Flexibelste Lösung.	Am langsamsten

Performance

```
$foreach = Measure-Command { foreach ($i in (1..1000000)) { $i * $i } }
$foreachmethod = Measure-Command { (1..1000000).ForEach{ $_ * $_ } }
$foreachobject = Measure-Command { (1..1000000) | ForEach-Object { $_ * $_ } }

"Foreach: $($foreach.TotalSeconds)"
"Foreach method: $($foreachmethod.TotalSeconds)"
"ForEach-Object: $($foreachobject.TotalSeconds)"

Example output:

Foreach: 1.9039875
Foreach method: 4.7559563
ForEach-Object: 10.7543821
```

Während `Foreach-Object` das langsamste ist, kann die Pipeline-Unterstützung nützlich sein, da Sie Elemente sofort beim Eintreffen verarbeiten können (beim Lesen einer Datei, Empfangen von Daten usw.). Dies kann sehr nützlich sein, wenn Sie mit Big Data und wenig Speicher arbeiten, da Sie nicht alle Daten vor der Verarbeitung in den Speicher laden müssen.

Examples

Zum

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

Eine typische Verwendung der for-Schleife ist die Bearbeitung einer Teilmenge der Werte in einem Array. In den meisten Fällen sollten Sie eine foreach -Anweisung verwenden, wenn Sie alle Werte in einem Array durchlaufen möchten.

Für jeden

`ForEach` hat in PowerShell zwei verschiedene Bedeutungen. Eines ist ein Schlüsselwort und das andere ist ein Alias für das Cmdlet ForEach-Object . Ersteres wird hier beschrieben.

In diesem Beispiel wird das Drucken aller Elemente in einem Array an den Konsolenhost veranschaulicht:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')
```

```
ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

Dieses Beispiel veranschaulicht das Erfassen der Ausgabe einer ForEach-Schleife:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternatively, Write-Output $Number
}
```

Wie das letzte Beispiel zeigt dieses Beispiel stattdessen das Erstellen eines Arrays vor dem Speichern der Schleife:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

Während

Eine while-Schleife wertet eine Bedingung aus, und bei true wird eine Aktion ausgeführt. Solange die Bedingung als wahr ausgewertet wird, wird die Aktion weiterhin ausgeführt.

```
while(condition){
  code_block
}
```

Im folgenden Beispiel wird eine Schleife erstellt, die von 10 auf 0 herunterzählt

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

Im Gegensatz zur Do während-Schleife wird die Bedingung vor der ersten Ausführung der Aktion ausgewertet. Die Aktion wird nicht ausgeführt, wenn die Anfangsbedingung falsch ist.

Hinweis: Wenn Sie die Bedingung bewerten, behandelt PowerShell das Vorhandensein eines Rückkehrobjekts als wahr. Dies kann auf verschiedene Arten verwendet werden. Nachfolgend finden Sie ein Beispiel für die Überwachung eines Prozesses. In diesem Beispiel wird ein Notepad-Prozess erstellt, und die aktuelle Shell wird in den Ruhezustand versetzt, solange dieser Prozess ausgeführt wird. Wenn Sie die Notepad-Instanz manuell schließen, schlägt die while-Bedingung fehl und die Schleife wird unterbrochen.

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
  Start-Sleep -Milliseconds 500
```

```
}
```

ForEach-Objekt

Das `ForEach-Object` funktioniert ähnlich wie die `foreach` Anweisung, übernimmt jedoch die Eingabe aus der Pipeline.

Grundlegende Verwendung

```
$object | ForEach-Object {
    code_block
}
```

Beispiel:

```
$names = @("Any","Bob","Celine","David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

`Foreach-Object` hat zwei Standard-Aliasnamen, `foreach` und `%` (Abkürzungssyntax). Am häufigsten ist `%` weil `foreach` mit der foreach-Anweisung verwechselt werden kann. Beispiele:

```
$names | % {
    "Hi, my name is $_!"
}

$names | foreach {
    "Hi, my name is $_!"
}
```

Erweiterte Nutzung

`Foreach-Object` sich von den alternativen `foreach` Lösungen, da es sich um ein Cmdlet handelt, das die Pipeline verwendet. Aus diesem Grund werden drei Skriptblöcke genauso wie ein Cmdlet oder eine erweiterte Funktion unterstützt:

- **Beginnen** : Wird einmal ausgeführt, bevor die Elemente, die von der Pipeline ankommen, durchlaufen werden. Wird normalerweise zum Erstellen von Funktionen für die Verwendung in der Schleife, zum Erstellen von Variablen, zum Öffnen von Verbindungen (Datenbank, Web +) usw. verwendet.
- **Prozess** : Wird einmal pro Artikel ausgeführt, der aus der Pipeline eingetroffen ist. "Normal" für jeden Codeblock. Dies ist die Standardeinstellung in den obigen Beispielen, wenn der Parameter nicht angegeben ist.
- **End** : Einmal ausgeführt, nachdem alle Elemente verarbeitet wurden. Wird normalerweise verwendet, um Verbindungen zu schließen, einen Bericht zu generieren usw.

Beispiel:

```
"Any","Bob","Celine","David" | ForEach-Object -Begin {
    $results = @()
} -Process {
    #Create and store message
    $results += "Hi, my name is $_!"
} -End {
    #Count messages and output
    Write-Host "Total messages: $($results.Count)"
    $results
}
```

Tun

Do-Loops sind nützlich, wenn Sie einen Codeblock immer mindestens einmal ausführen möchten. Eine Do-Schleife wertet die Bedingung nach der Ausführung des Codeblocks aus, im Gegensatz zu einer While-Schleife, die dies vor der Ausführung des Codeblocks ausführt.

Sie können do-loops auf zwei Arten verwenden:

- Schleife, *während* die Bedingung erfüllt ist:

  ```
  Do {
      code_block
  } while (condition)
  ```

- Schleife, *bis* die Bedingung wahr ist, mit anderen Worten, Schleife, während die Bedingung falsch ist:

  ```
  Do {
      code_block
  } until (condition)
  ```

Reale Beispiele:

```
$i = 0

Do {
    $i++
    "Number $i"
} while ($i -ne 3)

Do {
    $i++
    "Number $i"
} until ($i -eq 3)
```

Do-While und Do-Bis sind antonyme Schleifen. Wenn der Code gleich ist, wird die Bedingung umgekehrt. Das obige Beispiel veranschaulicht dieses Verhalten.

ForEach () - Methode

4,0

Anstelle des `ForEach-Object` besteht hier auch die Möglichkeit, eine `ForEach` Methode direkt auf Objektarrays zu verwenden

```
(1..10).ForEach({$_ * $_})
```

oder - falls gewünscht - können die Klammern um den Skriptblock weggelassen werden

```
(1..10).ForEach{$_ * $_}
```

Beides führt zur Ausgabe unten

```
1
4
9
16
25
36
49
64
81
100
```

Fortsetzen

Die `Continue` Operator arbeitet in `For` , `ForEach` , `While` und `Do` - Schleifen. Es überspringt die aktuelle Iteration der Schleife und springt an die Spitze der innersten Schleife.

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $I
}
```

Das obige wird 1 bis 20 auf der Konsole ausgeben, aber die Nummer 7 verpassen.

Hinweis : Wenn Sie eine Pipeline-Schleife verwenden, sollten Sie `return` anstelle von `Continue` .

Brechen

Der `break` eine Programmschleife sofort. Es kann in verwendet werden `For` , `ForEach` , `While` und `Do` oder in einem Schleifen - `Switch` Statement.

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

Das Obige zählt bis 15, hört jedoch auf, sobald 7 erreicht ist.

Hinweis : Wenn Sie eine Pipeline-Schleife verwenden, verhält sich `break` als `continue` . Zur Simulation `break` in der Pipeline - Schleife müssen Sie einige zusätzliche Logik integrieren, Cmdlets, etc. Es ist einfacher zu halten mit Nicht-Pipeline - Schleifen , wenn Sie verwenden müssen `break` .

Etiketten brechen

Break kann auch ein Label aufrufen, das vor der Instanziierung einer Schleife platziert wurde:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

Hinweis: Dieser Code erhöht `$i` auf `8` und `$j` auf `13` wodurch `$k` gleich `104` . Da `$k 100` überschreitet, wird der Code aus beiden Schleifen ausbrechen.

Schleifen online lesen: https://riptutorial.com/de/powershell/topic/1067/schleifen

Kapitel 57: SharePoint-Modul

Examples

Laden des SharePoint-Snap-Ins

Das Laden des SharePoint-Snapins kann folgendermaßen erfolgen:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Dies funktioniert nur in der 64-Bit-Version von PowerShell. Wenn im Fenster "Windows PowerShell (x86)" im Titel angezeigt wird, verwenden Sie die falsche Version.

Wenn das Snap-In bereits geladen ist, führt der obige Code zu einem Fehler. Die folgenden Elemente werden nur bei Bedarf geladen, was in Cmdlets / Funktionen verwendet werden kann:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Wenn Sie die SharePoint-Verwaltungsshell starten, wird alternativ das Snap-In automatisch eingefügt.

Führen Sie die folgenden Schritte aus, um eine Liste aller verfügbaren SharePoint-Cmdlets abzurufen:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Durchlaufen aller Listen einer Websitesammlung

Drucken Sie alle Listennamen und die Elementanzahl aus.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Rufen Sie alle installierten Funktionen einer Websitesammlung ab

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature kann auch im `-Web <WebUrl>` (`-Web <WebUrl>`), im `-Farm` (`-Farm`) und im Webanwendungsbereich (`-WebApplication <WebAppUrl>`) ausgeführt werden.

Holen Sie sich alle verwaisten Features einer Websitesammlung

Get-SPFeature kann auch verwendet werden, um alle Funktionen zu finden, die keinen Gültigkeitsbereich haben:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null )
```

SharePoint-Modul online lesen: https://riptutorial.com/de/powershell/topic/5147/sharepoint-modul

Kapitel 58: ShouldProcess verwenden

Syntax

- $ PSCmdlet.ShouldProcess ("Ziel")
- $ PSCmdlet.ShouldProcess ("Ziel", "Aktion")

Parameter

Parameter	Einzelheiten
Ziel	Die Ressource, die geändert wird
Aktion	Der Vorgang wird ausgeführt. Der Standardwert ist der Name des Cmdlets.

Bemerkungen

`$PSCmdlet.ShouldProcess()` schreibt automatisch eine Nachricht in die ausführliche Ausgabe.

```
PS> Invoke-MyCmdlet -Verbose
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Examples

Hinzufügen von -WhatIf und -Confirm Support zum Cmdlet

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Verwendung von ShouldProcess () mit einem Argument

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

Bei der Verwendung von `-WhatIf` :

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

Wenn Sie `-Confirm` :

```
Are you sure you want to perform this action?
Performing operation "Invoke-MyCmdlet" on target "Target of action"
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

Full Usage Beispiel

Andere Beispiele konnten mir nicht klar erklären, wie man die bedingte Logik auslöst.

Dieses Beispiel zeigt auch, dass die zugrunde liegenden Befehle auch das Flag -Confirm abhören.

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]
    param (
    [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]
    [string[]]$computerName,
    [parameter(Mandatory=$true)]
    [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,
    [boolean]$force = $false
)
BEGIN {
# translate action to numeric value required by the method
switch($action) {
    "Restart"
    {
        $_action = 2
        break
    }
    "LogOff"
    {
        $_action = 0
        break
    }
    "Shutdown"
    {
        $_action = 2
        break
    }
    "PowerOff"
    {
        $_action = 8
        break
    }
}
# to force, add 4 to the value
if($force)
{
    $_action += 4
}
write-verbose "Action set to $action"
}
PROCESS {
    write-verbose "Attempting to connect to $computername"
    # this is how we support -whatif and -confirm
    # which are enabled by the SupportsShouldProcess
```

```
    # parameter in the cmdlet bindnig
    if($pscmdlet.ShouldProcess($computername)) {
        get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -
name Win32Shutdown -argumentlist $_action
    }
}
}
#Usage:
#This will only output a description of the actions that this command would execute if -WhatIf
is removed.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

#This will request the permission of the caller to continue with this item.
#Attention: in this example you will get two confirmation request because all cmdlets called
by this cmdlet that also support ShouldProcess, will ask for their own confirmations...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm
```

ShouldProcess verwenden online lesen:
https://riptutorial.com/de/powershell/topic/1145/shouldprocess-verwenden

Kapitel 59: Sicherheit und Kryptographie

Examples

Berechnen der Hashcodes eines Strings über .Net Cryptography

`System.Security.Cryptography.HashAlgorithm` .Net `System.Security.Cryptography.HashAlgorithm` Namespace zum Generieren des Nachrichtenhashcodes mit den unterstützten Algorithmen.

```
$example="Nobody expects the Spanish Inquisition."

#calculate
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#convert to hex
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-
3D
```

Der `"sha256"` war der verwendete Hash-Algorithmus.

das `-` kann entfernt oder in Kleinschreibung geändert werden

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-","").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

Wenn das base64-Format bevorzugt wurde, verwenden Sie den base64-Konverter für die Ausgabe

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Sicherheit und Kryptographie online lesen:
https://riptutorial.com/de/powershell/topic/5683/sicherheit-und-kryptographie

Kapitel 60: Skripte signieren

Bemerkungen

Durch das Signieren eines Skripts werden Ihre Skripts mit allen Exeuktionsrichtlinien in PowerShell übereinstimmen und die Integrität eines Skripts sicherstellen. Signierte Skripts können nicht ausgeführt werden, wenn sie nach dem Signieren geändert wurden.

Die Skriptsignierung erfordert ein Codesignaturzertifikat. Empfehlungen:

- Persönliche Skripts / Tests (nicht freigegeben): Zertifikat einer vertrauenswürdigen Zertifizierungsstelle (intern oder von Drittanbietern) **ODER** ein selbstsigniertes Zertifikat.
- Freigegebene Organisation: Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle (intern oder von Drittanbietern)
- Freigegebene externe Organisation: Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle eines Dritten

Lesen Sie mehr bei about_Signing @ TechNet

Ausführungsrichtlinien

PowerShell verfügt über konfigurierbare Ausführungsrichtlinien, die festlegen, unter welchen Bedingungen ein Skript oder eine Konfiguration ausgeführt werden soll. Eine Ausführungsrichtlinie kann für mehrere Bereiche festgelegt werden. Computer, aktueller Benutzer und aktueller Prozess. **Ausführungsrichtlinien können leicht umgangen werden und sind nicht dazu gedacht, Benutzer einzuschränken, sondern sie vor unbeabsichtigten Verstößen gegen Signaturrichtlinien zu schützen.**

Die verfügbaren Richtlinien sind:

Rahmen	Beschreibung
Beschränkt	Keine Skripte erlaubt
AllSigned	Alle Skripte müssen signiert sein
RemoteSigned	Alle lokalen Skripte erlaubt; Nur signierte Remote-Skripte
Uneingeschränkt	Keine anforderungen Alle Skripts sind erlaubt, werden jedoch gewarnt, bevor Skripts ausgeführt werden, die aus dem Internet heruntergeladen wurden
Bypass	Alle Skripts sind zulässig und es werden keine Warnungen angezeigt
Nicht definiert	Entfernen Sie die aktuelle Ausführungsrichtlinie für den aktuellen Bereich. Verwendet die übergeordnete Richtlinie. Wenn alle Richtlinien undefiniert

Rahmen	Beschreibung
	sind, werden eingeschränkt verwendet.

Sie können die aktuellen Ausführungsrichtlinien mit `Set-ExecutionPolicy` -cmdlet, Gruppenrichtlinien oder dem Parameter `-ExecutionPolicy` , wenn Sie einen `powershell.exe` Prozess starten.

Weitere Informationen finden Sie unter about_Execution_Policies @ TechNet

Examples

Skript signieren

Das Signieren eines Skripts erfolgt mithilfe des `Set-AuthenticodeSignature` -cmdlet und eines Codesignaturzertifikats.

```
#Get the first available personal code-signing certificate for the logged on user
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#Sign script using certificate
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

Sie können ein Zertifikat auch aus einer `.pfx` Datei lesen:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

Das Skript ist gültig, bis das Zertifikat abläuft. Wenn Sie beim Signieren einen Zeitstempelserver verwenden, bleibt das Skript auch nach Ablauf des Zertifikats gültig. Es ist auch nützlich, die Vertrauenskette für das Zertifikat (einschließlich der Root-Berechtigung) hinzuzufügen, um den meisten Computern zu helfen, dem Zertifikat zu vertrauen, das zum Signieren des Skripts verwendet wird.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

Es wird empfohlen, einen Zeitstempelserver eines vertrauenswürdigen Zertifikatsanbieters wie Verisign, Comodo, Thawte usw. zu verwenden.

Ändern der Ausführungsrichtlinie mit Set-ExecutionPolicy

Verwenden Sie zum Ändern der Ausführungsrichtlinie für den Standardbereich (LocalMachine) Folgendes:

```
Set-ExecutionPolicy AllSigned
```

Verwenden Sie zum Ändern der Richtlinie für einen bestimmten Bereich Folgendes:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

Sie können die `-Force` unterdrücken, indem Sie den `-Force` Schalter hinzufügen.

Umgehung der Ausführungsrichtlinie für ein einzelnes Skript

Häufig müssen Sie möglicherweise ein nicht signiertes Skript ausführen, das nicht der aktuellen Ausführungsrichtlinie entspricht. Dies lässt sich leicht erreichen, indem die Ausführungsrichtlinie für diesen einzelnen Prozess umgangen wird. Beispiel:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Oder Sie können die Abkürzung verwenden:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Andere Ausführungsrichtlinien:

Politik	Beschreibung
`AllSigned`	Es können nur Skripts ausgeführt werden, die von einem vertrauenswürdigen Herausgeber signiert wurden.
`Bypass`	Keine Einschränkungen; Alle Windows PowerShell-Skripts können ausgeführt werden.
`Default`	Normalerweise `RemoteSigned` , wird aber über `RemoteSigned` gesteuert
`RemoteSigned`	Heruntergeladene Skripts müssen vor der Ausführung von einem vertrauenswürdigen Herausgeber signiert werden.
`Restricted`	Es können keine Skripts ausgeführt werden. Windows PowerShell kann nur im interaktiven Modus verwendet werden.
`Undefined`	N / A
`Unrestricted *`	Ähnlich wie `bypass`

`Unrestricted*` ***Vorsichtsmaßnahme:*** *Wenn Sie ein nicht signiertes Skript ausführen, das aus dem Internet heruntergeladen wurde, werden Sie vor der Ausführung zur Genehmigung aufgefordert.*

Weitere Informationen erhalten Sie hier .

Rufen Sie die aktuelle Ausführungsrichtlinie ab

Abrufen der effektiven Ausführungsrichtlinie für die aktuelle Sitzung:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Alle effektiven Ausführungsrichtlinien für die aktuelle Sitzung auflisten:

```
PS> Get-ExecutionPolicy -List

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

Listen Sie die Ausführungsrichtlinie für einen bestimmten Bereich auf, z. verarbeiten:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Signatur von einem signierten Skript abrufen

Rufen Sie mithilfe des `Get-AuthenticodeSignature` -cmdlet Informationen zur Authenticode-Signatur aus einem signierten Skript ab:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Erstellen eines selbstsignierten Codesignaturzertifikats zum Testen

Beim Signieren persönlicher Skripts oder beim Testen der Codesignierung kann es hilfreich sein, ein selbstsigniertes Codesignaturzertifikat zu erstellen.

5,0

Ab PowerShell 5.0 können Sie ein `New-SelfSignedCertificate` Codesignierungszertifikat mithilfe des `New-SelfSignedCertificate` -cmdlet generieren:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

In früheren Versionen können Sie ein selbstsigniertes Zertifikat mithilfe des Tools " `makecert.exe` im .NET Framework SDK und Windows SDK erstellen.

Ein selbstsigniertes Zertifikat wird nur von Computern als vertrauenswürdig eingestuft, auf denen das Zertifikat installiert ist. Für gemeinsam genutzte Skripts wird ein Zertifikat von einer

vertrauenswürdigen Zertifizierungsstelle (interner oder vertrauenswürdiger Drittanbieter) empfohlen.

Skripte signieren online lesen: https://riptutorial.com/de/powershell/topic/5670/skripte-signieren

Kapitel 61: Skriptvoraussetzungen erzwingen

Syntax

- #Requires -Version <N> [. <N>]
- # Erfordert -PSSnapin <PSSnapin-Name> [-Version <N> [. <N>]]
- #Requires -Modules {<Modulname> | <Hashtable>}
- #Requires –ShellId <ShellId>
- #ReAs -RunAsAdministrator erforderlich

Bemerkungen

`#requires` Anweisung `#requires` kann in jeder Zeile des Skripts platziert werden (es muss nicht die erste Zeile sein), muss aber die erste Anweisung in dieser Zeile sein.

In einem Skript können mehrere `#requires` Anweisungen verwendet werden.

Weitere Informationen finden Sie in der offiziellen Dokumentation zu Technet - about_about_Requires .

Examples

Erzwingen Sie die Mindestversion des Powershell-Hosts

```
#requires –version 4
```

Nachdem Sie versucht haben, dieses Skript in einer niedrigeren Version auszuführen, wird diese Fehlermeldung angezeigt

> . \ script.ps1: Das Skript 'script.ps1' kann nicht ausgeführt werden, da es in Zeile 1 für Windows PowerShell Version 5.0 die Anweisung "#requires" enthält. Die für das Skript erforderliche Version stimmt nicht mit der aktuell ausgeführten Version von Windows PowerShell Version 2.0 überein.

Erzwingen Sie die Ausführung des Skripts als Administrator

4,0

```
#requires –RunAsAdministrator
```

Nachdem Sie versucht haben, dieses Skript ohne Administratorrechte auszuführen, wird diese Fehlermeldung angezeigt

> . \ script.ps1: Das Skript 'script.ps1' kann nicht ausgeführt werden, da es eine "#requires" -Anweisung für die Ausführung als Administrator enthält. Die aktuelle

Windows PowerShell-Sitzung wird nicht als Administrator ausgeführt. Starten Sie Windows PowerShell mithilfe der Option Als Administrator ausführen, und versuchen Sie dann, das Skript erneut auszuführen.

Skriptvoraussetzungen erzwingen online lesen:
https://riptutorial.com/de/powershell/topic/5637/skriptvoraussetzungen-erzwingen

Kapitel 62: Spezielle Bediener

Examples

Array-Ausdrucksoperator

Gibt den Ausdruck als Array zurück.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Gibt ein Array mit einem Element zurück

```
@(Get-ChildItem $env:windir\System32)
```

Gibt ein Array mit allen Elementen im Ordner zurück (dies ist keine Verhaltensänderung des inneren Ausdrucks)

Operation aufrufen

```
$command = 'Get-ChildItem'
& $Command
```

Führt `Get-ChildItem`

Punktbeschaffungsoperator

. . \ myScript.ps1

`.\myScript.ps1` im aktuellen Bereich, sodass alle Funktionen und Variablen im aktuellen Bereich verfügbar sind.

Spezielle Bediener online lesen: https://riptutorial.com/de/powershell/topic/8981/spezielle-bediener

Kapitel 63: Spritzen

Einführung

Splatting ist eine Methode, mehrere Parameter als einzelne Einheit an einen Befehl zu übergeben. Dazu werden die Parameter und ihre Werte als Schlüssel / Wert-Paare in einer Hashtabelle gespeichert und mit dem Splatting-Operator `@` in ein Cmdlet aufgeteilt.

Splatting kann einen Befehl lesbarer machen und ermöglicht die Wiederverwendung von Parametern in mehreren Befehlsaufrufen.

Bemerkungen

Hinweis: Der Array-Ausdrucksoperator oder `@()` hat ein anderes Verhalten als der Splatting-Operator `@` .

Lesen Sie mehr bei about_Splatting @ TechNet

Examples

Splatting-Parameter

Das Aufteilen erfolgt durch Ersetzen des Dollarzeichens `$` durch den Aufteilungsoperator `@` wenn eine Variable verwendet wird, die eine HashTable mit Parametern und Werten in einem Befehlsaufruf enthält.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Ohne splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Sie können normale Parameter mit aufgeteilten Parametern kombinieren, um Ihren Aufrufen problemlos allgemeine Parameter hinzuzufügen.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Übergeben eines Switch-Parameters mithilfe von Splatting

So rufen Sie `Get-Process` mit dem Schalter `-FileVersionInfo` auf:

```
Get-Process -FileVersionInfo
```

Dies ist der Aufruf, der Splatting verwendet:

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Hinweis: Dies ist hilfreich, da Sie einen Standardsatz von Parametern erstellen und den Aufruf so oft ausführen können

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

Pfeifen und Spritzen

Das Deklarieren des Splats ist hilfreich, wenn Sie Parametersätze mehrmals oder mit geringfügigen Abweichungen verwenden möchten:

```
$splat = @{
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

Wenn der Splat jedoch nicht zur Wiederverwendung eingerückt ist, möchten Sie ihn möglicherweise nicht deklarieren. Es kann stattdessen geleitet werden:

```
@{
   ComputerName = $env:COMPUTERNAME
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
} | % { Get-WmiObject @_ }
```

Splatting von der Top-Level-Funktion zu einer Reihe von inneren Funktionen

Ohne Aufteilung ist es sehr umständlich, die Werte durch den Aufrufstapel weiterzuleiten. Wenn Sie jedoch die Aufteilung mit der Leistungsfähigkeit von **@PSBoundParameters kombinieren** , können Sie die Parameterauflistung der obersten Ebene durch die Ebenen **weiterleiten** .

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}


Outer-Method @parameters
```

Spritzen online lesen: https://riptutorial.com/de/powershell/topic/5647/spritzen

Kapitel 64: TCP-Kommunikation mit PowerShell

Examples

TCP-Listener

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Set up endpoint and start listening
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,$port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Wait for an incoming connection
            $data = $listener.AcceptTcpClient()

            # Stream setup
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Read data from stream and write it to host
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # Close TCP connection and stop listening
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Beginnen Sie mit dem Abhören und erfassen Sie eine beliebige Nachricht in der Variablen `$msg` :

```
$msg = Receive-TCPMessage -Port 29800
```

TCP Sender

```
Function Send-TCPMessage {
    Param (
```

```
            [Parameter(Mandatory=$true, Position=0)]
            [ValidateNotNullOrEmpty()]
            [string]
            $EndPoint
        ,
            [Parameter(Mandatory=$true, Position=1)]
            [int]
            $Port
        ,
            [Parameter(Mandatory=$true, Position=2)]
            [string]
            $Message
    )
    Process {
        # Setup connection
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # Setup stream wrtier
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Write message to stream
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Close connection and stream
        $Stream.Close()
        $Socket.Close()
    }
}
```

Nachricht senden mit:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Hinweis : TCP-Nachrichten werden möglicherweise von Ihrer Software-Firewall oder von externen Firewalls, die Sie durchqueren möchten, blockiert. Stellen Sie sicher, dass der TCP-Port, den Sie im obigen Befehl festgelegt haben, geöffnet ist und dass Sie den Listener auf demselben Port eingerichtet haben.

TCP-Kommunikation mit PowerShell online lesen:
https://riptutorial.com/de/powershell/topic/5125/tcp-kommunikation-mit-powershell

Kapitel 65: Umgang mit Geheimnissen und Anmeldeinformationen

Einführung

Um zu vermeiden, dass das Kennwort in *Klartext* gespeichert wird *, verwenden* wir in Powershell verschiedene Verschlüsselungsmethoden und speichern diese als sichere Zeichenfolge. Wenn Sie keinen Schlüssel oder einen sicheren Schlüssel angeben, funktioniert dies nur für denselben Benutzer auf demselben Computer. Er kann die verschlüsselte Zeichenfolge entschlüsseln, wenn Sie keine Schlüssel / Sicherungsschlüssel verwenden. Jeder Prozess, der unter demselben Benutzerkonto ausgeführt wird, kann diese verschlüsselte Zeichenfolge auf demselben Computer entschlüsseln.

Examples

Aufforderung zur Eingabe von Anmeldeinformationen

Um zur Eingabe von `Get-Credential` aufzufordern, sollten Sie fast immer das Cmdlet `Get-Credential` verwenden:

```
$credential = Get-Credential
```

Vorgefüllter Benutzername:

```
$credential = Get-Credential -UserName 'myUser'
```

Fügen Sie eine benutzerdefinierte Eingabeaufforderungsnachricht hinzu:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Zugriff auf das Klartext-Passwort

Das Kennwort in einem Berechtigungsnachweisobjekt ist verschlüsselt `[SecureString]` . Der einfachste Weg ist, ein `[NetworkCredential]` das das Passwort nicht verschlüsselt speichert:

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

Die `.GetNetworkCredential()` (`.GetNetworkCredential()`) ist nur für `[PSCredential]` -Objekte vorhanden.
Um direkt mit einem `[SecureString]` , verwenden Sie .NET-Methoden:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
```

```
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Mit gespeicherten Anmeldeinformationen arbeiten

Verwenden Sie zum einfachen Speichern und Abrufen verschlüsselter Anmeldeinformationen die integrierte XML-Serialisierung (Clixml) von PowerShell:

```
$credential = Get-Credential

$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

So importieren Sie erneut:

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

Zu beachten ist, dass standardmäßig die Windows-Datenschutz-API verwendet wird und der zum Verschlüsseln des Kennworts verwendete Schlüssel sowohl für den *Benutzer als auch für den Computer gilt* , auf dem der Code ausgeführt wird.

Daher kann der verschlüsselte Berechtigungsnachweis nicht von einem anderen Benutzer oder demselben Benutzer auf einem anderen Computer importiert werden.

Durch das Verschlüsseln mehrerer Versionen desselben Berechtigungsnachweises mit verschiedenen aktiven Benutzern und auf verschiedenen Computern können Sie das gleiche Geheimnis für mehrere Benutzer verfügbar machen.

Wenn Sie den Benutzernamen und den Computernamen in den Dateinamen eingeben, können Sie alle verschlüsselten Geheimnisse so speichern, dass derselbe Code ohne harte Kodierung verwendet werden kann:

Encrypter

```
# run as each user, and on each computer

$credential = Get-Credential

$credential | Export-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

Der Code, der die gespeicherten Anmeldeinformationen verwendet:

```
$credential = Import-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

Die korrekte Version der Datei für den laufenden Benutzer wird automatisch geladen (oder schlägt fehl, da die Datei nicht vorhanden ist).

Speichern der Anmeldeinformationen in verschlüsselter Form und Übergeben der Parameter als Parameter, wenn erforderlich

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# Now, $credObject is having the credentials stored and you can pass it wherever you want.


## Import Password with AES

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# Now, $credObject is having the credentials stored with AES Key and you can pass it wherever
you want.
```

Umgang mit Geheimnissen und Anmeldeinformationen online lesen:
https://riptutorial.com/de/powershell/topic/2917/umgang-mit-geheimnissen-und-anmeldeinformationen

Kapitel 66: Umgebungsvariablen

Examples

Windows-Umgebungsvariablen sind als PS-Laufwerk mit dem Namen Env sichtbar:

Sie können eine Liste mit allen Umgebungsvariablen anzeigen mit:
Get-Childitem env:

Sofortiger Aufruf von Umgebungsvariablen mit $ env:

```
$env:COMPUTERNAME
```

Umgebungsvariablen online lesen:
https://riptutorial.com/de/powershell/topic/5635/umgebungsvariablen

Kapitel 67: URL-Kodierung / Dekodierung

Bemerkungen

Der in den Beispielen für die *Decodierungs-URL* verwendete reguläre Ausdruck wurde aus RFC 2396, Anhang B: Analysieren einer URI-Referenz mit einem regulären Ausdruck entnommen. für die Nachwelt hier ein Zitat:

> Die folgende Zeile ist der reguläre Ausdruck zum Aufteilen einer URI-Referenz in ihre Komponenten.
>
> ```
> ^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
>  12            3  4          5       6  7        8 9
> ```
>
> Die Zahlen in der zweiten Zeile dienen nur zur besseren Lesbarkeit. Sie geben die Bezugspunkte für jeden Unterausdruck an (dh jede gepaarte Klammer). Wir bezeichnen den für den Teilausdruck passenden Wert als $. Zum Beispiel den obigen Ausdruck mit übereinstimmen
>
> ```
> http://www.ics.uci.edu/pub/ietf/uri/#Related
> ```
>
> führt zu folgenden Teilausdrücken:
>
> ```
> $1 = http:
> $2 = http
> $3 = //www.ics.uci.edu
> $4 = www.ics.uci.edu
> $5 = /pub/ietf/uri/
> $6 = <undefined>
> $7 = <undefined>
> $8 = #Related
> $9 = Related
> ```

Examples

Schnellstart: Kodierung

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility requires at least .NET 1.1 to be installed.
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Hinweis: Weitere Informationen zu HTTPUtility .

Schnellstart: Dekodierung

Hinweis: In diesen Beispielen werden die im Abschnitt *Schnellstart: Codierung* oben erstellten Variablen verwendet.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Returns: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Returns: http://test.com?test=my value

# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Returns: http://test.com?test=my+value

# Note: There is no `[uri]::UnescapeUriString()`;
#       which makes sense since the `[uri]::UnescapeDataString()`
#       function handles everything it would handle plus more.

# HttpUtility requires at least .NET 1.1 to be installed.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Returns: http://test.com?test=my value
```

Hinweis: Weitere Informationen zu HTTPUtility .

Encode Query String mit [uri] :: EscapeDataString () `

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Mit `[uri]::EscapeDataString()` werden Sie feststellen, dass der Apostroph (') nicht codiert wurde:

> https://example.vertigion.com/foos ? foo2 = komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & Komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

Encode-Abfragezeichenfolge mit [System.Web.HttpUtility] :: UrlEncode () `

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Mit `[System.Web.HttpUtility]::UrlEncode()` werden Sie feststellen, dass Leerzeichen in Pluszeichen (+) anstelle von `%20` :

> https://example.vertigion.com/foos ? foo2 = komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 & komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

URL mit `[uri] :: UnescapeDataString ()` decodieren

Kodiert mit `[uri]::EscapeDataString()`

Zuerst entschlüsseln wir die URL und den mit `[uri]::EscapeDataString()` codierten Abfrage-String im obigen Beispiel:

> https://example.vertigion.com/foos ? foo2 = komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & Komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2F

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
```

```
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Dies gibt Ihnen `[hashtable]$url_parts` ; Welches ist gleich (**Anmerkung:** die *Leerzeichen* in den komplexen Teilen sind *Leerzeichen*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Codiert mit `[System.Web.HttpUtility]::UrlEncode()`

Nun entschlüsseln wir die URL und den Abfrage-String, der mit `[System.Web.HttpUtility]::UrlEncode()` im obigen Beispiel codiert ist:

> https://example.vertigion.com/foos ? foo2 = komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 & komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
```

```
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Dies gibt Ihnen `[hashtable]$url_parts` , was gleich ist (**Hinweis:** Die *Leerzeichen* in den komplexen Teilen sind *Pluszeichen* (`+`) im ersten Teil und *Leerzeichen* im zweiten Teil):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

URL mit `[System.Web.HttpUtility] :: UrlDecode ()` decodieren

Kodiert mit `[uri]::EscapeDataString()`

Zuerst entschlüsseln wir die URL und den mit `[uri]::EscapeDataString()` codierten Abfrage-String im obigen Beispiel:

> https://example.vertigion.com/foos ? foo2 = komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 & Komplex% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks
```

```
if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Dies gibt Ihnen `[hashtable]$url_parts` ; Welches ist gleich (**Anmerkung:** die *Leerzeichen* in den komplexen Teilen sind *Leerzeichen*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Codiert mit `[System.Web.HttpUtility]::UrlEncode()`

Nun entschlüsseln wir die URL und den Abfrage-String, der mit `[System.Web.HttpUtility]::UrlEncode()` im obigen Beispiel codiert ist:

> https://example.vertigion.com/foos ? foo2 = komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 & komplex% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks
```

```
if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Dies gibt Ihnen `[hashtable]$url_parts` ; Welches ist gleich (**Anmerkung:** die *Leerzeichen* in den komplexen Teilen sind *Leerzeichen*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

URL-Kodierung / Dekodierung online lesen: https://riptutorial.com/de/powershell/topic/7352/url-kodierung---dekodierung

Kapitel 68: Variablen in PowerShell

Einführung

Variablen werden zum Speichern von Werten verwendet. Wenn der Wert von einem beliebigen Typ ist, müssen wir ihn irgendwo speichern, damit er in der Konsole / im Skript verwendet werden kann. Variablennamen in PowerShell beginnen wie in *$ Variable1* mit einem **$.** Werte werden mit **=** zugewiesen, wie **$ Variable1 = "Wert 1".** PowerShell unterstützt eine große Anzahl von Variablentypen. B. Textzeichenfolgen, Ganzzahlen, Dezimalzahlen, Arrays und sogar erweiterte Typen wie Versionsnummern oder IP-Adressen.

Examples

Einfache Variable

Alle Variablen in Powershell beginnen mit einem US-Dollar-Zeichen (`$`). Das einfachste Beispiel dafür ist:

```
$foo = "bar"
```

Diese Anweisung weist eine Variable namens `foo` mit dem Zeichenfolgenwert "bar" zu.

Eine Variable entfernen

Um eine Variable aus dem Speicher zu entfernen, können `Remove-Item` Cmdlet `Remove-Item` verwenden. Hinweis: Der Variablenname enthält NICHT das `$` .

```
Remove-Item Variable:\foo
```

`Variable` hat einen Anbieter, der den meisten * -item-Cmdlets erlaubt, ähnlich wie Dateisysteme zu arbeiten.

Eine andere Methode zum Entfernen von Variablen ist die Verwendung des Remove-Variable-Cmdlets und des Aliasnamens rv

```
$var = "Some Variable" #Define variable 'var' containing the string 'Some Variable'
$var #For test and show string 'Some Variable' on the console

Remove-Variable -Name var
$var

#also can use alias 'rv'
rv var
```

Umfang

Der Standardumfang für eine Variable ist der umschließende Behälter. Wenn Sie sich außerhalb eines Skripts oder eines anderen Containers befinden, ist der Gültigkeitsbereich `Global` . Um einen Bereich anzugeben, wird diesem der Variablenname `$scope:varname` vorangestellt:

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Ausgabe:

```
    Global Scope
    Function (local) scope
    Function (local) scope
    Global Scope
    Global Scope
```

Lesen einer CmdLet-Ausgabe

Standardmäßig gibt powershell die Ausgabe an die aufrufende Entität zurück. Betrachten Sie unter Beispiel,

```
Get-Process -Name excel
```

Dies würde einfach den laufenden Prozess, der dem Namen Excel entspricht, an die aufrufende Entität zurückgeben. In diesem Fall den PowerShell-Host. Es druckt etwas wie

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1037      54    67632      62544   617     5.23   4544   1 EXCEL
```

Wenn Sie nun die Ausgabe einer Variablen zuweisen, wird nichts gedruckt. Und natürlich enthält die Variable die Ausgabe. (Sei es ein String, Object - Jeder Typ für diese Angelegenheit)

```
$allExcel = Get-Process -Name excel
```

Nehmen wir an, Sie haben ein Szenario, in dem Sie eine Variable mit einem dynamischen Namen zuweisen möchten. Sie können den Parameter **`-OutVariable`**

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Beachten Sie, dass hier das '$' fehlt. Ein Hauptunterschied zwischen diesen beiden Zuweisungen

besteht darin, dass die Ausgabe außer der Zuweisung zu der Variablen AllRunningExcel auch gedruckt wird. Sie können sie auch einer anderen Variablen zuweisen.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Das obige Szenario ist jedoch sehr selten, beide Variablen $ VarOne und $ VarTwo haben den gleichen Wert.

Nun bedenke dies,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

Die erste Anweisung würde einfach Excel-Prozess abrufen und der MSOFFICE-Variablen zuweisen, und als Nächstes würden MS-Wortprozesse ausgeführt und an den vorhandenen Wert von MSOFFICE angehängt. Es würde so aussehen,

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1047      54    67720      64448   618     5.70   4544   1 EXCEL

   1172      70    50052      81780   584     1.83  14968   1 WINWORD
```

Listenzuordnung mehrerer Variablen

Powershell erlaubt die Mehrfachzuweisung von Variablen und behandelt fast alles wie ein Array oder eine Liste. Das bedeutet, anstatt etwas zu tun:

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

Sie können dies einfach tun:

```
$foo, $bar, $baz = $input.Split(".")
```

Da Powershell Zuweisungen auf diese Weise wie Listen behandelt, wird die letzte Variable zu einem Array der verbleibenden Werte, wenn sich in der Liste mehr Werte befinden als in der Liste der Variablen, denen sie zugewiesen werden sollen. Das bedeutet, dass Sie auch folgende Dinge tun können:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

Arrays

Die Array-Deklaration in Powershell ist fast identisch mit der Instanziierung aller anderen Variablen, dh Sie verwenden die Syntax `$name =` . Die Elemente im Array werden deklariert, indem sie durch Kommas (`,`) getrennt werden:

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Hinzufügen zu einer Arry

Das Hinzufügen eines Arrays ist so einfach wie der Operator `+`

```
$myArrayOfInts = $myArrayOfInts + 5
//now contains 1,2,3,4 & 5!
```

Arrays miteinander kombinieren

Dies ist wiederum so einfach wie der Operator `+`

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
//now 1,2,3,4,5,6,7
```

Variablen in PowerShell online lesen: https://riptutorial.com/de/powershell/topic/3457/variablen-in-powershell

Kapitel 69: Verhalten in PowerShell zurückgeben

Einführung

Es kann verwendet werden, um den aktuellen Bereich zu verlassen. Dies kann ein Funktions-, Skript- oder Skriptblock sein. In PowerShell wird das Ergebnis jeder Anweisung als Ausgabe zurückgegeben, auch ohne ein explizites Return-Schlüsselwort oder um anzuzeigen, dass das Ende des Bereichs erreicht wurde.

Bemerkungen

Weitere Informationen zur Rückkehrsemantik finden Sie auf der Seite about_Return in TechNet oder indem Sie `get-help return` von einer PowerShell-Eingabeaufforderung aufrufen.

Bemerkenswerte Q & A-Fragen mit weiteren Beispielen / Erklärungen:

- Funktionsrückgabewert in PowerShell
- PowerShell: Funktion hat keinen richtigen Rückgabewert

---

about_return auf MSDN erklärt es kurz und bündig:

> Das Return-Schlüsselwort beendet einen Funktions-, Skript- oder Skriptblock. Es kann verwendet werden, um einen Bereich an einem bestimmten Punkt zu verlassen, einen Wert zurückzugeben oder um anzuzeigen, dass das Ende des Bereichs erreicht wurde.
>
> Benutzer, die mit Sprachen wie C oder C # vertraut sind, möchten möglicherweise das Schlüsselwort Return verwenden, um die Logik des Verlassens eines Bereichs explizit zu machen.
>
> In Windows PowerShell werden die Ergebnisse jeder Anweisung als Ausgabe zurückgegeben, auch ohne eine Anweisung, die das Return-Schlüsselwort enthält. Sprachen wie C oder C # geben nur den oder die Werte zurück, die im Schlüsselwort Return angegeben sind.

Examples

Vorzeitige Ausfahrt

```
function earlyexit {
    "Hello"
    return
```

```
    "World"
}
```

"Hello" wird in die Ausgabepipeline eingefügt, "World" nicht

Erwischt! Rückkehr in die Pipeline

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

Pipeline-Cmdlets (z. `ForEach-Object` , `Where-Object` usw.) werden für Schließungen verwendet. Die Rückgabe hier wird nur zum nächsten Element in der Pipeline verschoben, nicht jedoch zum Beenden der Verarbeitung. Sie können **break** anstelle von **return verwenden,** wenn Sie die Verarbeitung beenden möchten.

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

Erwischt! Unerwünschte Ausgabe ignorieren

Inspiriert von

- PowerShell: Funktion hat keinen richtigen Rückgabewert

```
function bar {
 [System.Collections.ArrayList]$MyVariable = @()
 $MyVariable.Add("a") | Out-Null
 $MyVariable.Add("b") | Out-Null
 $MyVariable
}
```

`Out-Null` ist erforderlich, da die .NET `ArrayList.Add` Methode nach dem Hinzufügen die Anzahl der Elemente in der Auflistung zurückgibt. Wenn weggelassen, hätte die Pipeline `1, 2, "a", "b"`

Es gibt mehrere Möglichkeiten, unerwünschte Ausgaben wegzulassen:

```
function bar
{
    # New-Item cmdlet returns information about newly created file/folder
    New-Item "test1.txt" | out-null
    New-Item "test2.txt" > $null
    [void](New-Item "test3.txt")
    $tmp = New-Item "test4.txt"
}
```

Hinweis: Weitere Informationen dazu, warum `> $null` bevorzugt wird, finden Sie unter [Thema wurde noch nicht erstellt].

Rückgabe mit einem Wert

(paraphrased von about_return)

Die folgenden Methoden haben die gleichen Werte in der Pipeline

```
function foo {
    $a = "Hello"
    return $a
}

function bar {
    $a = "Hello"
    $a
    return
}

function quux {
    $a = "Hello"
    $a
}
```

Wie man mit Funktionen arbeitet, kehrt zurück

Eine Funktion gibt alles zurück, was nicht von etwas anderem erfasst wurde.
Wenn Sie das Schlüsselwort **return verwenden** , wird jede Anweisung nach der Zeile nicht ausgeführt.

So was:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Testfunktion
Wird zurückkehren:

- Start
- Der neu erstellte Registrierungsschlüssel (weil einige Anweisungen zum Erstellen von Ausgaben führen, die Sie möglicherweise nicht erwarten)
- Ja, es hat funktioniert!

Test-Function -ExceptionalReturn Gibt zurück:

- Start
- Verdammt, es hat nicht funktioniert!

Wenn Sie es so machen:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    . {
       "Start"
        if($ExceptionalReturn)
        {
            $Return = "Damn, it didn't work!"
            Return
        }
        New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
        $Return = "Yes, it worked!"
        Return
    } | Out-Null
    Return $Return
}
```

Testfunktion
Wird zurückkehren:

- Ja, es hat funktioniert!

Test-Function -ExceptionalReturn Gibt zurück:

- Verdammt, es hat nicht funktioniert!

Mit diesem Trick können Sie die zurückgegebene Ausgabe selbst dann steuern, wenn Sie nicht sicher sind, was die einzelnen Anweisungen aussagen werden.

So funktioniert es

```
.{<Statements>} | Out-Null
```

das . macht den folgenden Scriptblock im Code enthalten
das {} markiert den Skriptblock
die | Out-Null leitet alle unerwarteten Ausgaben an Out-Null (damit sie weg sind!).
Da der Skriptblock enthalten ist, erhält er denselben Bereich wie der Rest der Funktion.
So können Sie auf Variablen zugreifen, die innerhalb des Skriptblocks gemacht wurden.

Verhalten in PowerShell zurückgeben online lesen:
https://riptutorial.com/de/powershell/topic/4781/verhalten-in-powershell-zuruckgeben

Kapitel 70: Verwenden des Hilfesystems

Bemerkungen

`Get-Help` ist ein Cmdlet zum Lesen von Hilfethemen in PowerShell.

Lesen Sie mehr über ein TechNet

Examples

Aktualisieren des Hilfesystems

3,0

Ab PowerShell 3.0 können Sie die Offline-Hilfedokumentation mit einem einzigen Cmdlet herunterladen und aktualisieren.

```
Update-Help
```

So aktualisieren Sie die Hilfe auf mehreren Computern (oder Computern, die nicht mit dem Internet verbunden sind).

Führen Sie auf einem Computer die folgenden Hilfedateien aus

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Remote-Ausführung auf vielen Computern

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath
\\Server01\Share\Help -Credential $cred}
```

Get-Help verwenden

`Get-Help` kann zum Anzeigen der Hilfe in PowerShell verwendet werden. Sie können nach Cmdlets, Funktionen, Anbietern oder anderen Themen suchen.

Um die Hilfedokumentation zu Jobs anzuzeigen, verwenden Sie:

```
Get-Help about_Jobs
```

Sie können mithilfe von Platzhaltern nach Themen suchen. Wenn Sie verfügbare `about_` mit einem Titel `about_` , der mit `about_` , versuchen Sie `about_` :

```
Get-Help about_*
```

Wenn Sie Hilfe zu `Select-Object` , würden Sie Folgendes verwenden:

```
Get-Help Select-Object
```

Sie können auch die Aliase `help` oder `man` .

Online-Version eines Hilfethemas anzeigen

Sie können auf die Online-Hilfe zugreifen, indem Sie:

```
Get-Help Get-Command -Online
```

Beispiele ansehen

Verwendungsbeispiele für ein bestimmtes Cmdlet anzeigen.

```
Get-Help Get-Command -Examples
```

Anzeigen der vollständigen Hilfeseite

Sehen Sie sich die vollständige Dokumentation zum Thema an.

```
Get-Help Get-Command -Full
```

Anzeigen der Hilfe für einen bestimmten Parameter

Sie können die Hilfe für einen bestimmten Parameter anzeigen mit:

```
Get-Help Get-Content -Parameter Path
```

Verwenden des Hilfesystems online lesen:
https://riptutorial.com/de/powershell/topic/5644/verwenden-des-hilfesystems

Kapitel 71: Vorhandene statische Klassen verwenden

Einführung

Bei diesen Klassen handelt es sich um Referenzbibliotheken von Methoden und Eigenschaften, die ihren Zustand in einem Wort nicht unveränderlich ändern. Sie müssen sie nicht erstellen, Sie verwenden sie einfach. Klassen und Methoden wie diese werden als statische Klassen bezeichnet, da sie nicht erstellt, zerstört oder geändert werden. Sie können auf eine statische Klasse verweisen, indem Sie den Klassennamen mit eckigen Klammern umgeben.

Examples

Erstellen Sie sofort eine neue GUID

Verwenden Sie vorhandene .NET-Klassen sofort mit PowerShell, indem Sie [class] :: Method (args) verwenden:

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Auf ähnliche Weise können Sie in PowerShell 5+ das `New-Guid` Cmdlet verwenden:

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Um die GUID nur als `[String] .Guid` , referenziert die `.Guid` Eigenschaft:

```
[guid]::NewGuid().Guid
```

Verwenden der .Net Math-Klasse

Sie können die .Net Math-Klasse für Berechnungen verwenden ([System.Math]).

Wenn Sie wissen möchten, welche Methoden verfügbar sind, können Sie Folgendes verwenden:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Hier einige Beispiele zur Verwendung der Math-Klasse:

```
PS C:\> [System.Math]::Floor(9.42)
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

Typen hinzufügen

Fügen Sie unter Assemblyname die Bibliothek hinzu

```
Add-Type -AssemblyName "System.Math"
```

oder über Dateipfad:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Verwenden Sie den hinzugefügten Typ:

```
[CustomMath.NameSpace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Vorhandene statische Klassen verwenden online lesen:
https://riptutorial.com/de/powershell/topic/1522/vorhandene-statische-klassen-verwenden

Kapitel 72: Wie kann ich das neueste Artefakt von Artifactory mithilfe des Powershell-Skripts (Version 2.0 oder darunter) herunterladen?

Einführung

In dieser Dokumentation werden Schritte zum Herunterladen des neuesten Artefakts aus einem JFrog Artifactory-Repository mit Powershell-Skript (Version 2.0 oder niedriger) erläutert.

Examples

Powershell-Skript zum Herunterladen des neuesten artifcat

```
$username = 'user'
$password= 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')

[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloaUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifcatUrlResponse=$client.DownloadString($downloaUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifcatUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Wie kann ich das neueste Artefakt von Artifactory mithilfe des Powershell-Skripts (Version 2.0 oder darunter) herunterladen? online lesen: https://riptutorial.com/de/powershell/topic/8883/wie-kann-ich-das-neueste-artefakt-von-artifactory-mithilfe-des-powershell-skripts--version-2-0-oder-darunter--herunterladen-

Kapitel 73: WMI und CIM

Bemerkungen

CIM vs. WMI

Ab PowerShell 3.0 gibt es zwei Möglichkeiten, mit Verwaltungsklassen in PowerShell, WMI und CIM zu arbeiten. PowerShell 1.0 und 2.0 unterstützten nur das WMI-Modul, das nun durch das neue und verbesserte CIM-Modul ersetzt wird. In einer späteren Version von PowerShell werden die WMI-Cmdlets entfernt.

Vergleich von CIM- und WMI-Modulen:

CIM-Cmdlet	WMI-Cmdlet	Was es macht
Get-CimInstance	Get-WmiObject	Ruft CIM / WMI-Objekte für eine Klasse ab
Aufruf-CimMethod	Aufruf-WmiMethod	Ruft eine CIM / WMI-Klassenmethode auf
Register-CimIndicationEvent	Register-WmiEvent	Registriert ein Ereignis für eine CIM / WMI-Klasse
Remove-CimInstance	Remove-WmiObject	CIM / WMI-Objekt entfernen
Set-CimInstance	Set-WmiInstance	Updates / Speichert CIM / WMI-Objekt
Get-CimAssociatedInstance	N / A	Verbundene Instanzen (verknüpfte Objekte / Klassen) abrufen
Get-CimClass	Get-WmiObject -List	Listet CIM / WMI-Klassen auf
New-CimInstance	N / A	Neues CIM-Objekt erstellen
Get-CimSession	N / A	Listet CIM-Sitzungen auf
New-CimSession	N / A	Neue CIM-Sitzung erstellen
New-CimSessionOption	N / A	Erzeugt ein Objekt mit Sitzungsoptionen. Protokoll, Verschlüsselung, Deaktivierung der Verschlüsselung usw. (zur Verwendung mit `New-`

CIM-Cmdlet	WMI-Cmdlet	Was es macht
		`CimSession`)
Remove-CimSession	N / A	Entfernt / stoppt die CIM-Sitzung

Zusätzliche Ressourcen

Sollte ich CIM oder WMI mit Windows PowerShell verwenden? @ Hey, Scripting Guy! Blog

Examples

Objekte abfragen

CIM / WMI wird am häufigsten verwendet, um Informationen oder Konfigurationen auf einem Gerät abzufragen. Es gibt eine Klasse, die eine Konfiguration, einen Prozess, einen Benutzer usw. darstellt. In PowerShell gibt es mehrere Möglichkeiten, auf diese Klassen und Instanzen zuzugreifen. Am häufigsten werden jedoch die Cmdlets `Get-CimInstance` (CIM) oder `Get-WmiObject` (WMI) verwendet.

Alle Objekte für die CIM-Klasse auflisten

Sie können alle Instanzen einer Klasse auflisten.

3,0

CIM:

```
> Get-CimInstance -ClassName Win32_Process

ProcessId Name                         HandleCount WorkingSetSize VirtualSize
--------- ----                         ----------- -------------- -----------
0         System Idle Process          0           4096           65536
4         System                       1459        32768          3563520
480       Secure System                0           3731456        0
484       smss.exe                     52          372736         2199029891072
....
....
```

WMI:

```
Get-WmiObject -Class Win32_Process
```

Filter verwenden

Sie können einen Filter anwenden, um nur bestimmte Instanzen einer CIM / WMI-Klasse abzurufen. Filter werden mit `WQL` (Standard) oder CQL (Hinzufügen `-QueryDialect CQL`) geschrieben. `-Filter` verwendet den `WHERE` Teil einer vollständigen WQL / CQL-Abfrage.

3,0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 676         88305664       2199697199104
```

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"

...
Caption                     : powershell.exe
CommandLine                 : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName           : Win32_Process
CreationDate                : 20160913184324.393887+120
CSCreationClassName         : Win32_ComputerSystem
CSName                      : STACKOVERFLOW-PC
Description                 : powershell.exe
ExecutablePath              : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState              :
Handle                      : 4800
HandleCount                 : 673
....
```

Verwendung einer WQL-Abfrage:

Sie können auch eine WQL / CQL-Abfrage verwenden, um Instanzen abzufragen und zu filtern.

3,0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 673         88387584       2199696674816
```

Abfragen von Objekten in einem anderen Namespace:

3,0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct


displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct

__GENUS                   : 2
__CLASS                   : AntiVirusProduct
__SUPERCLASS              :
__DYNASTY                 : AntiVirusProduct
__RELPATH                 : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}"
__PROPERTY_COUNT          : 6
__DERIVATION              : {}
__SERVER                  : STACKOVERFLOW-PC
__NAMESPACE               : ROOT\SecurityCenter2
__PATH                    : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            : STACKOVERFLOW-PC
```

Klassen und Namespaces

In CIM und WMI stehen viele Klassen zur Verfügung, die in mehrere Namespaces unterteilt sind. Der gebräuchlichste (und Standard-) Namespace in Windows ist `root/cimv2` . Um die Klasse zu finden, kann es nützlich sein, alle aufzulisten oder zu suchen.

Liste der verfügbaren Klassen

Sie können alle verfügbaren Klassen im Standardnamespace (`root/cimv2`) auf einem Computer `root/cimv2` .

3,0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Suche nach einer Klasse

Sie können mithilfe von Platzhaltern nach bestimmten Klassen suchen. Ex: Finden Klassen mit dem Wort `process` .

3,0

CIM:

```
> Get-CimClass -ClassName "*Process*"

   NameSpace: ROOT/CIMV2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------

Win32_ProcessTrace                  {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace             {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace              {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                         {}                   {Caption, Description, InstallDate,
Name...}
Win32_Process                       {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor                       {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor                     {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

Listen Sie Klassen in einem anderen Namespace auf

Der Root-Namespace wird einfach als `root` . Sie können Klassen in einem anderen Namespace mit dem Parameter `-NameSpace` .

3,0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"

   NameSpace: ROOT/SecurityCenter2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------
....
AntiSpywareProduct                  {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct                    {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
FirewallProduct                     {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

Liste der verfügbaren Namespaces

Um verfügbare `__NAMESPACE` Namespaces von `root` (oder einem anderen Namespace) zu finden, `__NAMESPACE` Sie die Objekte in der `__NAMESPACE` Klasse für diesen Namespace ab.

3,0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"

Name            PSComputerName
----            --------------
subscription
DEFAULT
CIMV2
msdtc
Cli
SECURITY
HyperVCluster
SecurityCenter2
RSOP
PEH
StandardCimv2
WMI
directory
Policy
virtualization
Interop
Hardware
ServiceModel
SecurityCenter
Microsoft
aspnet
Appv
```

WMI:

```
Get-WmiObject -List -Namespace "root"
```

WMI und CIM online lesen: https://riptutorial.com/de/powershell/topic/6808/wmi-und-cim

Kapitel 74: Zeichenketten

Syntax

- "(Doppelte Anführungszeichen) Zeichenfolge"
- 'Wörtliche Zeichenfolge'
- @ "
  Here-String
  "@
- @ '
  Wörtliche Here-Zeichenfolge
  '@

Bemerkungen

Strings sind Objekte, die Text darstellen.

Examples

Erstellen einer Basiszeichenfolge

String

Zeichenfolgen werden erstellt, indem der Text in doppelte Anführungszeichen gesetzt wird. Strings in doppelten Anführungszeichen können Variablen und Sonderzeichen auswerten.

```
$myString = "Some basic text"
$mySecondString = "String with a $variable"
```

Um ein doppeltes Anführungszeichen in einer Zeichenfolge zu verwenden, muss es mit dem Escape-Zeichen, backtick (`), maskiert werden. Einzelne Anführungszeichen können innerhalb einer doppelten Anführungszeichenfolge verwendet werden.

```
$myString = "A `"double quoted`" string which also has 'single quotes'."
```

Wörtliche Zeichenfolge

Literalzeichenfolgen sind Zeichenfolgen, die keine Variablen und Sonderzeichen auswerten. Es wird mit einfachen Anführungszeichen erstellt.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

Verwenden Sie doppelte einfache Anführungszeichen oder eine Literal-Here-Zeichenfolge, um einfache Anführungszeichen innerhalb einer Literal-Zeichenfolge zu verwenden. Doppelte Qutos können sicher innerhalb einer Zeichenfolge verwendet werden

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Zeichenfolge formatieren

```
$hash = @{ city = 'Berlin' }

$result = 'You should really visit {0}' -f $hash.city
Write-Host $result #prints "You should really visit Berlin"
```

Formatzeichenfolgen können mit dem Operator `-f` oder der statischen .NET-Methode `[String]::Format(string format, args)` verwendet werden.

Mehrzeilige Zeichenfolge

Es gibt mehrere Möglichkeiten, eine mehrzeilige Zeichenfolge in PowerShell zu erstellen:

- Sie können die Sonderzeichen für Wagenrücklauf und / oder Zeilenvorschub manuell verwenden oder die `NewLine` -environment-Variable verwenden, um den Systemwert "Newline" einzufügen.

  ```
  "Hello`r`nWorld"
  "Hello{0}World" -f [environment]::NewLine
  ```

- Erstellen Sie einen Zeilenumbruch beim Definieren einer Zeichenfolge (vor dem Schließen des Zitats)

  ```
  "Hello
  World"
  ```

- Verwenden Sie hier eine Zeichenfolge. *Dies ist die gebräuchlichste Technik.*

  ```
  @"
  Hello
  World
  "@
  ```

Here-String

Here-Strings sind sehr nützlich, wenn Sie mehrzeilige Strings erstellen. Einer der größten Vorteile im Vergleich zu anderen mehrzeiligen Zeichenketten besteht darin, dass Sie Anführungszeichen verwenden können, ohne sie mit einem Backtick zu schützen.

Here-String

Here-Strings beginnen mit `@"` und einem Zeilenumbruch und enden mit `"@` in der eigenen Zeile (`"@` **muss die ersten Zeichen in der Zeile sein, nicht einmal Leerzeichen / Tabulator**).

```
@"
Simple
    Multiline string
with "quotes"
"@
```

Wörtliche Here-Zeichenfolge

Sie können auch einen Literal-Here-String mit einfachen Anführungszeichen erstellen, wenn Sie nicht möchten, dass Ausdrücke wie ein normaler Literal-String erweitert werden.

```
@'
The following line won't be expanded
$(Get-Date)
because this is a literal here-string
'@
```

Zeichenketten verketten

Verwenden von Variablen in einer Zeichenfolge

Sie können Zeichenfolgen mithilfe von Variablen in doppelten Anführungszeichenfolgen verketten. Dies funktioniert nicht mit Eigenschaften.

```
$string1 = "Power"
$string2 = "Shell"
"Greetings from $string1$string2"
```

Verwenden Sie den Operator `+`

Sie können Zeichenfolgen auch mit dem Operator `+` verknüpfen.

```
$string1 = "Greetings from"
$string2 = "PowerShell"
$string1 + " " + $string2
```

Dies funktioniert auch mit Eigenschaften von Objekten.

```
"The title of this console is '" + $host.Name + "'"
```

Unterausdrücke verwenden

Die Ausgabe / das Ergebnis eines Unterausdrucks `$()` kann in einer Zeichenfolge verwendet werden. Dies ist nützlich, wenn Sie auf Eigenschaften eines Objekts zugreifen oder einen komplexen Ausdruck ausführen möchten. Unterausdrücke können mehrere durch Semikolon getrennte Anweisungen enthalten `;`

```
"Tomorrow is $((Get-Date).AddDays(1).DayOfWeek)"
```

Spezielle Charaktere

Bei Verwendung in einer Zeichenfolge mit doppelten Anführungszeichen stellt das Escape-Zeichen (backtick `` ` ``) ein Sonderzeichen dar.

```
`0    #Null
`a    #Alert/Beep
`b    #Backspace
`f    #Form feed (used for printer output)
`n    #New line
`r    #Carriage return
`t    #Horizontal tab
`v    #Vertical tab (used for printer output)
```

Beispiel:

```
> "This`tuses`ttab`r`nThis is on a second line"
This    uses    tab
This is on a second line
```

Sie können auch Sonderzeichen mit speziellen Bedeutungen umgehen:

```
`#    #Comment-operator
`$    #Variable operator
``    #Escape character
`'    #Single quote
`"    #Double quote
```

Zeichenketten online lesen: https://riptutorial.com/de/powershell/topic/5124/zeichenketten

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit PowerShell	4444, autosvet, Brant Bobby, Chris N, Clijsters, Community, DarkLite1, DAXaholic, Eitan, FoxDeploy, Gordon Bell, Greg Bray, Ian Miller, It-Z, JNYRanger, Jonas, Luboš Turek, Mark Wragg, Mathieu Buisson, Mrk, Nacimota, oɔɯəɹ, Poorkenny, Sam Martin, th1rdey3, TheIncorrigible1, Tim, tjrobinson, TravisEz13, vonPryz, Xalorous
2	ActiveDirectory-Modul	Lachie White
3	Aliase	jumbo
4	Amazon Web Services (AWS) - einfacher Speicherdienst (S3)	Trevor Sullivan
5	Anonymisieren Sie IP (v4 und v6) in einer Textdatei mit Powershell	NooJ
6	Anweisung wechseln	Anthony Neace, Frode F., jumbo, oɔɯəɹ, Ranadip Dutta, TravisEz13
7	Arbeiten mit der PowerShell-Pipeline	Alban, Atsch, Clijsters, Deptor, James Ruskin, Keith, o ɔɯəɹ, Sam Martin
8	Archivmodul	James Ruskin, RapidCoder
9	Ausführbare Dateien ausführen	RamenChef, W1M0R
10	Automatische Variablen	Brant Bobby, jumbo, Mateusz Piotrowski, Moerwald, Ranadip Dutta, Roman
11	Automatische Variablen - Teil 2	Roman
12	Bedingte Logik	Liam, lloyd, miken32, TravisEz13
13	Berechnete Eigenschaften	Prageeth Saravanan
14	Cmdlet-Benennung	TravisEz13

15	CSV-Analyse	Andrei Epure, Frode F.
16	Dynamische PowerShell-Parameter	Poorkenny
17	Einbetten von verwaltetem Code (C # | VB)	ajb101
18	Einführung in Pester	Frode F., Sam Martin
19	Einführung in Psake	Roman
20	Eingebaute Variablen	Trevor Sullivan
21	Email schicken	Adam M., jimmyb, megamorf, NooJ, Ranadip Dutta, void, Yusuke Arakawa
22	Erkennung von Amazon Web Services (AWS)	Trevor Sullivan
23	Erstellen von klassenbasierten DSC-Ressourcen	Trevor Sullivan
24	Fehlerbehandlung	Prageeth Saravanan
25	Fortschrittsbalken verwenden	Clijsters, jumbo, Ranadip Dutta
26	Gemeinsame Parameter	autosvet, jumbo, RamenChef
27	Geplantes Aufgabenmodul	Sam Martin
28	Gewünschter Zustand Konfiguration	autosvet, CmdrTchort, Frode F., RamenChef
29	Grundlegende Vorgänge	Euro Micelli, Ranadip Dutta, TravisEz13
30	GUI in Powershell	Sam Martin
31	HashTables	Florian Meyer, Ranadip Dutta, TravisEz13
32	Infrastrukturautomatisierung	Giulio Caccin, Ranadip Dutta
33	ISE-Modul	Florian Meyer
34	Kommentarbasierte Hilfe	Christophe
35	Kommunikation mit RESTful-APIs	autosvet, Clijsters, HAL9256, kdtong, RamenChef, Ranadip Dutta, Sam Martin, YChi Lu

36	Mit Objekten arbeiten	Chris N, djwork, Mathieu Buisson, megamorf
37	Mit XML-Dateien arbeiten	autosvet, Frode F., Giorgio Gambino, Lieven Keersmaekers, RamenChef, Richard, Rowshi
38	Module, Skripte und Funktionen	Frode F., Ranadip Dutta, Xalorous
39	MongoDB	Thomas Gerot, Zteffer
40	Operatoren	Anthony Neace, Bevo, Clijsters, Gordon Bell, JPBlanc, Mark Wragg, Ranadip Dutta
41	Paketverwaltung	TravisEz13
42	Parametersätze	Bert Levrau, Poorkenny
43	PowerShell "Streams"; Debuggen, ausführlich, Warnung, Fehler, Ausgabe und Informationen	DarkLite1, Dave Anderson, megamorf
44	Powershell SQL-Abfragen	Venkatakrishnan
45	PowerShell.exe Befehlszeile	Frode F.
46	PowerShell-Funktionen	Bert Levrau, Eris, James Ruskin, Luke Ryan, niksofteng, Ranadip Dutta, Richard, TessellatingHeckler, TravisEz13, Xalorous
47	PowerShell-Hintergrundjobs	Clijsters, mattnicola, Ranadip Dutta, Richard, TravisEz13
48	PowerShell-Klassen	boeprox, Brant Bobby, Frode F., Jaqueline Vanek, Mert Gülsoy, Ranadip Dutta, xvorsx
49	Powershell-Module	autosvet, Mike Shepard, TravisEz13, Trevor Sullivan
50	Powershell-Profile	Frode F., Kolob Canyon
51	Powershell-Remoting	Avshalom, megamorf, Moerwald, Sam Martin, ShaneC
52	PowerShell-Workflows	Trevor Sullivan
53	PSScriptAnalyzer - PowerShell Script Analyzer	Mark Wragg, mattnicola
54	Regeln der Namensgebung	niksofteng
55	Reguläre Ausdrücke	Frode F.

56	Schleifen	Blockhead, Christopher G. Lewis, Clijsters, CmdrTchort, DAXaholic, Eris, Frode F., Gomibushi, Gordon Bell, Jay Bazuzi, Jon, jumbo, mákos, Poorkenny, Ranadip Dutta, Richard, Roman, SeeuD1, Shawn Esterman, StephenP, TessellatingHeckler, TheIncorrigible1, VertigoRay
57	SharePoint-Modul	Raziel
58	ShouldProcess verwenden	Brant Bobby, Charlie Joynt, Schwarzie2478
59	Sicherheit und Kryptographie	YChi Lu
60	Skripte signieren	AP., Frode F.
61	Skriptvoraussetzungen erzwingen	autosvet, Frode F., jumbo, RamenChef
62	Spezielle Bediener	TravisEz13
63	Spritzen	autosvet, Frode F., Moerwald, Petru Zaharia, Poorkenny, RamenChef, Ranadip Dutta, TravisEz13, xXhRQ8sD2L7Z
64	TCP-Kommunikation mit PowerShell	autosvet, RamenChef, Richard
65	Umgang mit Geheimnissen und Anmeldeinformationen	4444, briantist, Ranadip Dutta, TravisEz13
66	Umgebungsvariablen	autosvet
67	URL-Kodierung / Dekodierung	VertigoRay
68	Variablen in PowerShell	autosvet, Eris, Liam, Prageeth Saravanan, Ranadip Dutta , restless1987, Steve K
69	Verhalten in PowerShell zurückgeben	Bert Levrau, camilohe, Eris, jumbo, Ranadip Dutta, Thomas Gerot
70	Verwenden des Hilfesystems	Frode F., Madniz, mattnicola, RamenChef
71	Vorhandene statische Klassen verwenden	Austin T French, briantist, motcke, Ranadip Dutta, Xenophane
72	Wie kann ich das neueste Artefakt von Artifactory mithilfe des Powershell-	ANIL

	Skripts (Version 2.0 oder darunter) herunterladen?	
73	WMI und CIM	Frode F.
74	Zeichenketten	Frode F., restless1987, void