

Tobias Weltner

Windows PowerShell 2.0 Scripting für Administratoren

Das Praxis-Kochbuch zur automatisierten Verwaltung
von Windows-Systemen



Fachbibliothek

Microsoft
Press

Dr. Tobias Weltner

Windows PowerShell 2.0 Scripting für Administratoren

Microsoft
Press

Dr. Tobias Weltner: Windows PowerShell 2.0 Scripting für Administratoren
Microsoft Press Deutschland, Konrad-Zuse-Str. 1, 85716 Unterschleißheim
Copyright © 2011 Microsoft Press Deutschland

Das in diesem Buch enthaltene Programmmaterial ist mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor, Übersetzer und der Verlag übernehmen folglich keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programmmaterials oder Teilen davon entsteht. Die in diesem Buch erwähnten Software- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Marken und unterliegen als solche den gesetzlichen Bestimmungen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller.

Das Werk einschließlich aller Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die in den Beispielen verwendeten Namen von Firmen, Organisationen, Produkten, Domänen, Personen, Orten, Ereignissen sowie E-Mail-Adressen und Logos sind frei erfunden, soweit nichts anderes angegeben ist. Jede Ähnlichkeit mit tatsächlichen Firmen, Organisationen, Produkten, Domänen, Personen, Orten, Ereignissen, E-Mail-Adressen und Logos ist rein zufällig.

Kommentare und Fragen können Sie gerne an uns richten:
Microsoft Press Deutschland
Konrad-Zuse-Straße 1
85716 Unterschleißheim

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
13 12 11

ISBN 978-3-86645-5-680-8, PDF-eBook-ISBN 978-3-86645-351-7

© 2011 O'Reilly Verlag GmbH & Co. KG
Balthasarstr. 81, 50670 Köln
Alle Rechte vorbehalten

Fachlektorat: Georg Weiherer, Münzenberg
Korrektorat: mediaService, Siegen (www.media-service.tv)
Layout und Satz: Gerhard Alfes, mediaService, Siegen (www.media-service.tv)
Umschlaggestaltung: Hommer Design GmbH, Haar (www.HommerDesign.com)
Gesamtherstellung: Kösel, Krugzell (www.KoeselBuch.de)

Inhaltsverzeichnis

Einleitung	13
1 Texte und Textauswertungen	17
Text erstellen	18
Text vom Benutzer erfragen	22
Text über ein Dialogfeld erfragen	23
Text aus einer Datei lesen	24
Zeilenumbruch oder Anführungszeichen in Texten	28
Informationen in einen Text einfügen	29
Dynamische Informationen formatieren	33
Stichwörter in Texten finden	39
Informationen aus einem Text extrahieren	45
Textinhalte durch andere Texte ersetzen	51
Leerzeichen aus Texten entfernen	53
Doppelte Wörter entfernen	55
Wortbereiche finden	56
Text in Groß- oder Kleinbuchstaben umwandeln	58
Texte in einzelne Zeichen umwandeln	58
Texte aus mehreren Einzeltexten zusammensetzen	60
Texte in eine Datei schreiben	62
Einträge eines Logbuchs zeilenweise auswerten	64
Zusammenfassung	69
2 Datum und Zeit	71
Aktuelles Datum ermitteln	72
Kalenderwoche oder Wochentag bestimmen	72
Dateiname mit Zeitstempel erstellen	79
Deutsches Datumsformat verarbeiten	80
Internationale Datumsformate verarbeiten	80
Kulturneutrale Datumsformate verarbeiten	83
Frei definierbare Datumsformate verarbeiten	85
Datum auf Gültigkeit prüfen	86
Zeitdifferenz ermitteln	87
WMI-Zeitangaben umwandeln	90
Systemticks umwandeln	91
Zusammenfassung	92

3 Listen und Arrays	95
Neues Array anlegen	96
Mehrdimensionale Arrays verwenden	100
Auf Arrayelemente zugreifen	101
Alle Arrayelemente der Reihe nach bearbeiten	103
Arrayinhalte sortieren	106
Prüfen, ob ein Array ein bestimmtes Element enthält	107
Arrayelemente nachträglich hinzufügen und entfernen	109
Mehrere Arrays zusammenfassen	112
Arrays miteinander vergleichen	113
Schlüssel-Wert-Paare speichern	115
Schlüssel-Wert-Paare sortieren	117
Zusammenfassung	119
4 PowerShell-Pipeline	121
Informationen filtern	123
Objekteigenschaften (Spalten) auswählen	127
Ergebnisse einzeln verarbeiten	129
Ausgaben sortieren	133
Ergebnisse gruppieren	136
Ergebnisse in Text umwandeln	141
Zusammenfassung	143
5 Bedingungen und Schleifen	145
Entscheidungen fällen	146
Mehrere Vergleiche kombinieren	150
Arrayinhalte vergleichen	152
Code ausführen, wenn Bedingung erfüllt ist	153
Elemente der Pipeline einzeln bearbeiten	159
Code mit Schleifen mehrfach wiederholen	160
Mehrere Ergebnisse bearbeiten	164
Zusammenfassung	168
6 Skripts, Funktionen und Fehlerbehandlung	171
Skripts verfassen	172
Skripts starten	174
Skripts automatisch ausführen	176
Skripts in der Pipeline verwenden	177
Rückgabewert eines Skripts festlegen	180
Optionale Parameter definieren	181
Zwingend erforderliche Parameter definieren	183
Switch-Parameter definieren	184
Parameter validieren	186

Mehrere Werte pro Parameter übergeben	187
Hilfe integrieren	190
Neue Funktion verfassen	192
Funktion löschen	194
Funktion mit Schreibschutz versehen	195
Pipeline-Filter anlegen	196
Feststellen, ob ein Fehler aufgetreten ist	197
Fehler in Skripts und Funktionen abfangen	200
Eigene Fehler auslösen	203
Zusammenfassung	204
7 Dateisystem	207
Inhalt eines Ordners auflisten	208
Dateien finden, die einem Kriterium entsprechen	211
Relative Pfadnamen verwenden	213
Relativen Pfad auflösen	214
Prüfen, ob ein Pfad Platzhalter enthält	217
Pfadnamen konstruieren	218
Pfadbestandteile auswerten	219
Systempfad ermitteln	220
Dateien kopieren	224
Ordner kopieren	225
Neue Ordner anlegen	226
Neue Datei anlegen	227
Text in eine Datei schreiben	230
Text aus einer Datei lesen	232
Inhalt einer Textdatei löschen	233
Datei löschen	234
Datei oder Ordner umbenennen	235
Auf erweiterte Dateieigenschaften zugreifen	237
Dateiattribute lesen und ändern	240
Dateien mit einem bestimmten Attribut finden	242
Dateinamen mit speziellen Zeichen verarbeiten	244
Aktuellen Ordner bestimmen (oder setzen)	246
Pfadnamen auflösen	248
Veränderungen an Ordnerinhalten per Snapshot auswerten	250
Echtzeit-Überwachung von Änderungen an Ordnern	252
Echtzeit-Überwachung von Änderungen an Dateien	257
Ungültige Dateinamen ermitteln	260
Zusammenfassung	260
8 Registrierungsdatenbank	263
Alle Unterschlüssel eines Schlüssels lesen	265
Neuen Registrierungsschlüssel anlegen	268

Prüfen, ob ein Schlüssel existiert	271
Bestimmten Schlüssel suchen	272
Schlüssel löschen	273
Wert eines Schlüssels lesen	274
Standardwert eines Schlüssels lesen	278
Werte vieler Schlüssel auslesen	278
Prüfen, ob ein bestimmter Wert existiert	279
Wert eines Schlüssels ändern	280
Wert eines Schlüssels löschen	282
Standardwert eines Schlüssels löschen	284
Windows-Registrierungs-Editor öffnen	285
In der Registrierungsdatenbank navigieren	286
Andere Orte der Registrierungsdatenbank ansprechen	288
Remote auf Registrierungsdatenbank zugreifen	289
Zusammenfassung	291
9 Prozesse und Anwendungen	293
Laufende Prozesse sichtbar machen	294
Feststellen, ob ein Prozess läuft	297
Anzahl der Instanzen eines Prozesses bestimmen	298
Prozess starten	299
Prozess unter anderer Identität starten	301
Prozess als Administrator starten	304
Prozess beenden	305
Abgestürzte Prozesse finden und beenden	307
Abkürzungen für häufige Befehle einrichten	307
Ausgaben eines Programms weiterverarbeiten	309
Zusammenfassung	312
10 Dienste	315
Alle Dienste auflisten	316
Bestimmte Dienste finden	318
Dienste starten oder stoppen	321
Abhängige Dienste finden	323
Dienste-Einstellungen ändern	324
Auf eine Dienststatusänderung warten	326
Uptime eines Diensts bestimmen	327
Zugrunde liegende Dienstprogramme ermitteln	329
Dienste-Informationen als CSV in Excel importieren	330
Dienstzustand überprüfen	331
Neuen Dienst installieren	332
Dienst entfernen	332
Zusammenfassung	333

11 Ereignisprotokoll	335
Alle Ereignisprotokolle auflisten	336
Prüfen, ob ein Ereignisprotokoll existiert	338
Neueste Ereignisse auflisten	338
Ereignisse mit einer bestimmten ID finden	342
Fehlermeldungen im Ereignisprotokoll finden	349
Nach einem Stichwort suchen	350
Ereignisse nach Häufigkeit gruppieren	351
Neue Einträge in ein Ereignisprotokoll schreiben	353
Ereignisprotokoll archivieren	354
EVT-Dateien einlesen	356
Inhalt eines Ereignisprotokolls löschen	357
Remotezugriff auf Ereignisprotokolle	357
Ereignisprotokoll konfigurieren	358
Zusammenfassung	361
12 Zugriffsberechtigungen	363
Zugriffsberechtigungen lesen	364
Zugriffsrechte für Dateien und Ordner festlegen	367
Zugriffsrechte eines Registrierungsschlüssels festlegen	374
Besitz übernehmen	377
Zusammenfassung	379
13 Digitale Zertifikate und Sicherheit	381
Skriptausführung mit <i>ExecutionPolicy</i> erlauben	382
Installiertes Codesigning-Zertifikat auswählen	384
Neues Zertifikat aus Datei installieren	386
Zertifikat aus einer Datei lesen	388
Selbstsigniertes Testzertifikat erstellen	389
Prüfen, ob ein Zertifikat vertrauenswürdig ist	393
Selbstsigniertes Zertifikat für vertrauenswürdig erklären	394
PowerShell-Skript signieren	396
Skriptsignatur überprüfen	399
Zertifikat als <i>.cer</i> -Datei exportieren	401
Zertifikat als <i>.pfx</i> -Datei exportieren	402
Zertifikat löschen	403
Zusammenfassung	405
14 XML-Daten verarbeiten	407
Auf den Inhalt einer XML-Datei zugreifen	408
Mit XPath Informationen in XML finden	413
XML-Informationen filtern	417
Neues XML-Dokument erstellen	418

Neues XML-Dokument aus Schablone erstellen	420
Inhalt einer XML-Datei ändern	426
Zusammenfassung	428
15 Mit Datenbanken arbeiten	429
Datenbankunterstützung testen	430
Über .NET Framework auf Datenbanken zugreifen	431
Mit Datasets Ergebnisdaten verarbeiten	441
XML mit SQL aus Microsoft SQL Server abrufen	443
Über COM auf eine Datenbank zugreifen	445
Datenbankinhalte ändern	446
Verfügbare SQL Server-Instanzen ermitteln	448
Zusammenfassung	449
16 Benutzerverwaltung und Active Directory	451
Active Directory-Cmdlets verwenden	453
Mit Active Directory verbinden	456
Neues Benutzerkonto anlegen	458
Viele neue Benutzerkonten anlegen	460
Benutzerkonto löschen	462
Kennwort eines Benutzerkontos ändern	463
Benutzerkonto suchen	464
Mit LDAP-Filtern suchen	466
Neue Organisationseinheit anlegen	468
Eigenschaften eines Benutzerkontos lesen	469
Eigenschaften eines Benutzerkontos ändern	471
Gruppenmitgliedschaft eines Benutzerkontos auflisten	473
Gruppenmitglieder auslesen	474
Benutzerkonto zu einer Gruppe hinzufügen	474
Benutzerkonto aus einer Gruppe entfernen	475
Auf ein lokales Benutzerkonto zugreifen	476
Auf eine lokale Gruppe zugreifen	477
Lokale Benutzerkonten finden	479
Alle lokalen Gruppen auflisten	480
Prüfen, ob ein lokales Konto existiert	482
Neues lokales Benutzerkonto anlegen	483
Benutzerkonto in eine lokale Gruppe aufnehmen	485
Benutzerkonto aus einer lokalen Gruppe entfernen	486
Lokales Benutzerkonto konfigurieren	487
Prüfen, ob ein lokales Benutzerkonto Mitglied einer Gruppe ist	488
Lokales Benutzerkonto aktivieren oder deaktivieren	489
Lokales Benutzerkonto löschen	490
Zusammenfassung	492

17 PowerShell-Snap-Ins und Module	493
Feststellen, welche Erweiterungen verfügbar sind	494
PowerShell-Erweiterungen laden	496
Neue Cmdlets einer Erweiterung sichtbar machen	497
Neue Provider einer Erweiterung sichtbar machen	499
Eigene PowerShell-Erweiterungen herstellen	502
Zusammenfassung	505
18 PowerShell Remoting	507
Cmdlets mit eigener Remoteunterstützung finden	508
PowerShell Remoting einrichten	510
Interaktiv auf ein Remotesystem zugreifen	512
Befehl oder Skript remote ausführen	513
Berechtigungen für PowerShell Remoting konfigurieren	518
Autostart-Skripts für Remoteverbindungen einrichten	519
PSSession öffnen und daraus Befehle importieren	521
Remotebefehle als Befehlserweiterung nutzen	523
Remoteauthentifizierung mit CredSSP einrichten	525
Zusammenfassung	528
19 Hintergrundjobs	531
Cmdlets mit eigener Parallelverarbeitung finden	532
Aufgaben im Hintergrund ausführen	536
Aufgaben regelmäßig im Hintergrund ausführen	538
Anmeldeinformationen in Hintergrundjobs verwenden	540
Hintergrundjobs überwachen	541
Zusammenfassung	543
Stichwortverzeichnis	545
Der Autor	559

Einleitung

»Those who forget to script are doomed to repeat their work«

Haben Sie auch schon einmal in einer riesigen Protokolldatei versucht, die für Sie wichtigen Informationen herauszufischen? Oder mussten Sie zig Benutzerkonten editieren und haben sich mit der Maus dabei beinahe ein Karpaltunnelsyndrom eingehandelt?

Während Maus und grafische Oberfläche für Privatanwender oder für gelegentliche Ausnahmefälle eine gute Idee ist, greifen IT-Profis gern zu Automationswerkzeugen und Skriptsprachen, um Routineaufgaben schnell und in gleichbleibender Qualität zu lösen. Auf Windows-basierten Systemen setzt sich dazu immer stärker die Skriptsprache Windows PowerShell durch und löst ältere Skriptsprachen wie VBScript zunehmend ab.

Anders als bei den meisten älteren Skriptsprachen gibt es für PowerShell kaum Einsatzgrenzen. PowerShell wird mit über 250 vielseitigen Befehlen ausgeliefert, den *Cmdlets*. Weitere lassen sich über sogenannte *Module* nachrüsten. Microsoft bietet Erweiterungen für die Verwaltung von Microsoft Exchange, SQL Server, Active Directory, Gruppenrichtlinien und vielen anderen Produkten an, und Drittanbieter wie VMware unterstützen ihre Produkte ebenfalls durch PowerShell-Erweiterungen. Wem Cmdlets und virtuelle Laufwerke nicht genügen, der kann sich selbst helfen und mit Windows PowerShell auf genau dieselben Systemfunktionen zugreifen, die bisher nur gestandenen Anwendungsentwicklern offen standen, und daraus eigene PowerShell-Erweiterungen schmieden.

Wie das alles konkret funktioniert und wie man mit Windows PowerShell die eine oder andere Aufgabe tatsächlich löst, beantwortet dieses Buch. Es versteht sich als Lösungssammlung für Hunderte von Routineproblemen, organisiert in 19 Themenbereiche, und versetzt Sie in die Lage, ohne große Suchaktionen schnell die entscheidenden Codebeispiele für Ihre Aufgaben zu finden.

Obwohl die meisten Lösungen nur aus wenigen Zeilen PowerShell-Code bestehen, finden Sie alle Codebeispiele außerdem in Form einer Textdatei, sodass Sie sie bequem kopieren und auf Ihrem System testen und einsetzen können. Sie finden die Codebeispiele zum Download unter der folgenden URL:

www.microsoft-press.de/support.asp?s110=680 oder msp.oreilly.de/support/9783866456808/551

Dieses Buch ist also ein Nachschlagewerk für die PowerShell-Praxis. Es ist kein Lehrbuch, das Ihnen die Sprache »PowerShell« in allen Einzelheiten nahebringen möchte. Wollen Sie kleinere Probleme lösen, können Sie die vielen Codebeispiele in diesem Buch ohne größeres Vorwissen sofort einsetzen. Wer umfangreichere Projekte mit PowerShell umsetzen möchte, sollte sich parallel den bei Microsoft Press erschienenen Titel »Scripting mit Windows PowerShell 2.0 – Schritt für Schritt – Der Einsteiger-Workshop« (ISBN: 978-3-86645-669-3) anschauen. Dieses Buch bildet die ideale Ergänzung zum vorliegenden Werk und legt die sprachlichen Grundlagen, um zum PowerShell-Profi zu werden.

Natürlich ist mir bewusst, wie viele Menschen an der Entstehung eines solchen Buchs beteiligt sind. Deshalb möchte ich an dieser Stelle dem Verlag danken, insbesondere meinem Lektor Florian Helmchen und meinem technischen Lektor Georg Weiherer, die beide mit großer

Leidenschaft und Sorgfalt dieses Buch mitgestaltet haben. Ein ganz herzlicher Dank geht auch an meine Kollegen aus der Microsoft MVP-Newsgruppe für ihre Unterstützung.

PowerShell nachrüsten

Setzen Sie Windows 7 oder Windows Server 2008 R2 ein, ist Windows PowerShell 2.0 bereits fester Bestandteil des Betriebssystems und Sie können sofort damit arbeiten. Auf älteren Windows-Betriebssystemen bis hinab zu Windows XP kann Windows PowerShell kostenlos nachgerüstet werden. Da es für die unterschiedlichen Betriebssysteme auch unterschiedliche Updatepakete gibt, verzichte ich hier auf eine lange Liste kryptischer Downloadlinks. Besuchen Sie stattdessen eine Internetsuchseite und geben Sie als Suchbegriff »KB968930« sowie Ihr Betriebssystem an, also beispielsweise »KB968930 Windows XP«. Dies führt Sie zuverlässig zu der entsprechenden Downloadseite von Microsoft.

Achten Sie bitte darauf, nicht versehentlich die (veraltete) Version 1.0 der PowerShell zu installieren. Welche Version der PowerShell bei Ihnen zur Verfügung steht, erkennen Sie am einfachsten an der Copyrightmeldung der PowerShell-Konsole: Steht hier ein Copyright aus 2009, nutzen Sie die aktuelle Version 2.0. Meldet der Copyrighthinweis das Jahr 2006, verwenden Sie die veraltete Version 1.0 und sollten schnellstmöglich auf PowerShell 2.0 aufrüsten.

Lassen Sie sich übrigens nicht vom Pfadnamen des Verzeichnisses irritieren, in dem PowerShell installiert wird. Dieses Verzeichnis heißt immer `%WINDIR%\System32\Windows-PowerShell\v1.0`, gleichgültig welche PowerShell-Version Sie installiert haben, und ist ein gutes Beispiel dafür, dass es keine clevere Idee war, Versionsnamen in Produktpfadnamen zu integrieren – aus Gründen der Abwärtskompatibilität ließen sich diese nämlich mit Einführung der Version 2.0 nicht mehr verändern.

Damit bleibt mir jetzt, Ihnen viel Spaß und Erfolg mit Windows PowerShell und den Lösungen aus diesem Buch zu wünschen. Gern können Sie mich unter tobias.weltner@email.de kontaktieren, wenn Sie Feedback zum Buch oder Anregungen an mich schicken möchten oder vielleicht in Ihrer Firma eine Inhouse-Schulung zu Windows PowerShell organisieren wollen. Unter <http://www.powershell.com> beantworte ich zudem in den kostenlosen (englischsprachigen) Foren gern Fragen rund um Windows PowerShell.

Dr. Tobias Weltner
Hannover, im März 2011

Kapitel 1

Texte und Textauswertungen

In diesem Kapitel:

Text erstellen	18
Text vom Benutzer erfragen	22
Text über ein Dialogfeld erfragen	23
Text aus einer Datei lesen	24
Zeilenumbruch oder Anführungszeichen in Texten	28
Informationen in einen Text einfügen	29
Dynamische Informationen formatieren	33
Stichwörter in Texten finden	39
Informationen aus einem Text extrahieren	45
Textinhalte durch andere Texte ersetzen	51
Leerzeichen aus Texten entfernen	53
Doppelte Wörter entfernen	55
Wortbereiche finden	56
Text in Groß- oder Kleinbuchstaben umwandeln	58
Texte in einzelne Zeichen umwandeln	58

Texte aus mehreren Einzeltexten zusammensetzen	60
Texte in eine Datei schreiben	62
Einträge eines Logbuchs zeilenweise auswerten	64
Zusammenfassung	69

Obwohl PowerShell eine »objektorientierte« Umgebung ist, spielen simple Textinformationen in der Administration und Automation noch immer eine überragende Rolle. Häufig möchte man Textinformationen vom Benutzer erfragen, diese Informationen vielleicht überprüfen, wertvolle Informationen aus Textdateien und Logbüchern filtern oder Texte als neue Datei speichern. Zudem liegen Textinformationen häufig nicht von vornherein im gewünschten Format vor, sodass eine weitere Routineaufgabe darin besteht, Informationen aus einzelnen Texten zu extrahieren. Aus einem Pfadnamen möchte man beispielsweise nur den Dateinamen oder die Dateierweiterung extrahieren.

In diesem Kapitel erfahren Sie, wie PowerShell Texte verarbeitet werden und wie Sie in Textinformationen Stichwörter finden, Ersetzungen vornehmen oder aus mehreren Textteilen einen Gesamttext zusammenfügen. Sie erfahren auch, wie Texte aus Dateien gelesen oder in neue Dateien geschrieben beziehungsweise angehängt werden.

Text erstellen

Sie wollen Text in einer Variablen speichern, zum Beispiel, um den Text später mehrfach zu verwenden oder als Argument für andere Befehle zu nutzen.

Lösung

Setzen Sie den Text in einfache Anführungszeichen und weisen Sie den Text einer Variablen zu:

```
PS > $text = 'Beispieltext'  
PS > $text  
Beispieltext
```

Falls Sie im Text Variablen auflösen oder Sonderzeichen einfügen möchten, greifen Sie zu doppelten Anführungszeichen:

```
PS > $text = "Die Variable `$env:windir` enthält: `$env:windir`"  
PS > $text  
Die Variable $env:windir enthält:  
C:\Windows
```

»\$« löst die nachfolgende Variable auf, ersetzt den Variablennamen also durch den Inhalt der Variable. In doppelten Anführungszeichen kommt also den Sonderzeichen »`« und »\$« eine besondere Bedeutung zu.

»`« kann das nachfolgende Zeichen entweder entwerfen (wie beispielsweise in »`\$, um innerhalb doppelter Anführungszeichen ein Dollarzeichen darzustellen, das andernfalls als Variable fehlinterpretiert werden könnte) oder Sonderzeichen wie Zeilenumbrüche einfügen (wie in »`n«). Dieses Zeichen wird auch »Backtick« genannt.

HINWEIS Sie erhalten das Sonderzeichen auf deutschen Tastaturen, indem Sie die -Taste festhalten, die Taste rechts neben  drücken und dann die -Taste drücken.

Möchten Sie Anführungszeichen innerhalb des Texts einfügen, geben Sie jeweils zwei Anführungszeichen hintereinander an:

```
PS > $text = 'Hallo ''Welt'''
```

```
PS > $text
```

```
Hallo 'Welt'
```

```
PS > $text = "Hallo ""Welt"""
```

```
PS > $text
```

```
Hallo "Welt"
```

```
PS >
```

Mehrzeilige Texte lassen sich mit *Here-Strings* zuweisen:

```
PS > $text = @"
```

```
>> Here-Strings umfassen sämtlichen Text,
```

```
>> den Sie zwischen den Anfangs- und End-Kennzeichen eingeben.
```

```
>> Der Text schließt dabei auch Zeilenumbrüche ein.
```

```
>> "@
```

```
>>
```

```
PS > $text
```

```
Here-Strings umfassen sämtlichen Text,
```

```
den Sie zwischen den Anfangs- und End-Kennzeichen eingeben.
```

```
Der Text schließt dabei auch Zeilenumbrüche ein.
```

Here-Strings beginnen mit »@"« oder »@'«. Danach muss unmittelbar ein Zeilenumbruch folgen. Der *Here-String* wird mit dem korrespondierenden »"@« beziehungsweise »'@« abgeschlossen. Auch diese Zeichen müssen in einer eigenen Zeile stehen. Die folgende Zuweisung ist deshalb ungültig:

```
PS > $text = @"Here-Strings umfassen sämtlichen Text,
```

```
Unbekanntes Token im Quelltext.
```

```
Bei Zeile:1 Zeichen:9
```

```
+ $text = <<<< @"Here-Strings umfassen sämtlichen Text,
```

```
+ CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
```

```
+ FullyQualifiedErrorId : UnrecognizedToken
```

Hintergrund

Texte müssen bei PowerShell nicht immer in Anführungszeichen gesetzt werden. Sofern der Text keine Leer- und Sonderzeichen enthält, darf er als Parameter für Cmdlets auch ohne Anführungszeichen verwendet werden:

```
PS > Write-Host Hallo  
Hallo
```

Das allerdings ist nur eine Ausnahme, damit Konsolenbefehle im Alltag schnell und unkompliziert funktionieren. Die beiden folgenden Befehle liefern dasselbe Resultat, nämlich ein Ordnerlisting des Windows-Ordners:

```
PS > dir C:\windows  
PS > dir 'C:\windows'
```

Spätestens wenn Texte nicht als Argument für Befehle verwendet werden, sondern zum Beispiel einer Variablen zugewiesen werden, sind Anführungszeichen aber Pflicht:




```
PS > # falsch:  
PS > $a = Hallo  
Die Benennung "Hallo" wurde nicht als Cmdlet, Funktion, ausführbares Programm oder  
Skriptdatei erkannt. Überprüfen Sie die Benennung, und versuchen Sie es erneut.  
Bei Zeile:1 Zeichen:10  
+ $a = Hallo <<<<  
PS > # richtig:  
PS > $a = 'Hallo'  
  
PS > # falsch:  
PS > $host.UI.Write(Hallo)  
Fehlende ")" im Methodenaufruf.  
Bei Zeile:1 Zeichen:16  
+ $host.UI.Write(H <<<< allo)  
PS > # richtig:  
PS > $host.UI.Write('Hallo')  
Hallo
```

Die Anführungszeichen sorgen hier also für klare Verhältnisse, weil PowerShell ohne sie nicht wissen kann, ob Sie einen Text oder vielleicht einen Befehl meinen, der so heißt wie Ihr Text.

Zur Textbegrenzung dürfen Sie einfache (»'«) oder doppelte (»"«) Anführungszeichen verwenden, solange Sie sich einheitlich für eines der beiden Zeichen entscheiden. Die folgende Mischform ist nicht erlaubt:

```
PS > "Hallo"  
>>
```

Tatsächlich finden Sie sich im letzten Beispiel anschließend im Vervollständigungsmodus wieder, den PowerShell immer aktiviert, wenn eine Anweisung noch nicht komplett ist, und den Sie

an der besonderen Eingabeaufforderung »>>>« erkennen. In diesem Beispiel wartet PowerShell auf die abschließenden doppelten Anführungszeichen. Geben Sie also entweder doppelte Anführungszeichen ein und drücken die -Taste oder brechen mit + ab.

Stellen Sie einen Text in einfache Anführungszeichen, wird der Text genauso wie angegeben (wörtlich) übernommen. Sonderzeichen (wie zum Beispiel das Sonderzeichen für einen Zeilenumbruch) werden in diesem Fall also wörtlich verstanden und ausgegeben, führen also zu keinem Zeilenumbruch:

```
PS > $text = 'Dieser Text besteht aus zwei Zeilen.`nDies ist die zweite Zeile.'  
PS > $text  
Dieser Text besteht aus zwei Zeilen.`nDies ist die zweite Zeile.
```

Doppelte Anführungszeichen kennzeichnen Text, in dem PowerShell automatisch Ersetzungen durchführt. Ersetzt werden Sonderzeichen, die mit einem Backtick-Zeichen eingeleitet werden, sowie Variablen, die mit einem `$`-Zeichen beginnen:

```
PS > "Der Windows-Ordner liegt hier: `"$env:windir`""  
Der Windows-Ordner liegt hier: "C:\Windows"
```

HINWEIS

Stellen Sie einen Text in doppelte Anführungszeichen, werden darin zwar Variablen automatisch aufgelöst und durch ihren Inhalt ersetzt. Allerdings löst PowerShell tatsächlich stets nur die Variable selbst auf. Befindet sich in der Variablen ein Objekt und möchten Sie auf eine Objekteigenschaft zugreifen, würde dies nicht funktionieren:

```
PS > $weihnachten = New-Timespan -End 24.12.2011  
PS > $weihnachten.Days  
313  
PS > "Es sind noch $weihnachten.Days Tage bis Weihnachten"  
Es sind noch 313.14:15:10.2829705.Days Tage bis Weihnachten
```

Tatsächlich hätte PowerShell hier nur den Inhalt von `$weihnachten` aufgelöst, und weil es sich um ein komplexes Objekt handelt, dessen (unleserliche) Textrepräsentation angezeigt. Daran angehängt finden Sie bei näherem Hinsehen `.Days`. Die Objekteigenschaft wurde also nicht aufgelöst. In diesem Fall verwenden Sie entweder Direktvariablen, die mit runden Klammern genau anzeigen, welcher Codeteil aufgelöst werden soll:

```
PS > "Es sind noch $($weihnachten.Days) Tage bis Weihnachten"  
Es sind noch 313 Tage bis Weihnachten
```

Oder Sie greifen zum Formatierungsoperator `-f`, der an anderer Stelle in diesem Buch ausführlich beschrieben wird und in eine Textschablone dynamische Informationen einfügt:

```
PS > 'Es sind noch {0} Tage bis Weihnachten' -f $weihnachten.Days  
Es sind noch 313 Tage bis Weihnachten
```

Text vom Benutzer erfragen

Sie möchten Text vom Benutzer interaktiv erfragen.

Lösung


Verwenden Sie *Read-Host* und geben Sie eine Frage an:

```
PS > Read-Host 'Ihr Geburtstag'
Ihr Geburtstag: 18.3.1912
18.3.1912
```

Auf diese Weise könnten Sie beispielsweise ein Datum erfragen und die Zeitdifferenz berechnen:

```
PS > $datum = Read-Host 'Ihr Geburtstag'
Ihr Geburtstag: 18.3.1912
PS > $differenz = New-Timespan $datum
PS > $differenz
Days                : 35092
Hours               : 11
Minutes             : 14
(...)
TotalMinutes        : 50533154,58922
TotalSeconds        : 3031989275,3532
TotalMilliseconds   : 3031989275353,2
PS > $tage = $differenz.TotalDays
PS > "Sie haben bereits $tage Tage erlebt!"
Sie haben bereits 35092.4684647361 Tage erlebt!
PS > "Sie haben bereits $($differenz.TotalDays) Tage erlebt!"
Sie haben bereits 35092.4684647361 Tage erlebt!
PS > 'Sie haben bereits {0:0} Tage erlebt!' -f $differenz.TotalDays
Sie haben bereits 35092 Tage erlebt!
```

Hintergrund

Read-Host liest interaktive Eingaben vom Benutzer so lange, bis dieser die -Taste drückt. Damit der Anwender weiß, wonach gefragt wird, kann eine Frage (ein »Prompt«) festgelegt werden. *Read-Host* fügt an diesen Prompt immer automatisch einen Doppelpunkt an. Das Ergebnis der Eingabe kann in einer Variable gespeichert und danach von anderen Befehlen weiterverarbeitet werden.

Voraussetzung dafür ist natürlich, dass der Anwender auch tatsächlich die erwarteten Eingaben tätigt. Im vorangegangenen Beispiel wurde ein Datum erwartet. Gibt der Benutzer kein gültiges Datum ein, führt der Code zu einem Fehler. Um solche Probleme zu vermeiden, sollten Sie die Eingabe zuerst auf Gültigkeit überprüfen. Wie Sie Eingaben auf Gültigkeit prüfen, erfahren Sie in einem anderen Kapitel in diesem Buch.

Die Ausgabe der mit *New-Timespan* berechneten Zeitdifferenz erfolgt im Beispiel auf drei unterschiedliche Arten. Im ersten Fall wird die Information *TotalDays* zuerst in einer Variablen namens *\$tage* gespeichert. Diese Variable kann dann in einem Text automatisch aufgelöst werden, wenn Sie dabei doppelte Anführungszeichen verwenden.

Im zweiten Fall wird die gewünschte Information *TotalDays* direkt ausgelesen, also nicht zuerst in einer eigenen Variablen gespeichert. In diesem Fall muss der gesamte Ausdruck innerhalb des Texts in die Konstruktion *\$(...)* gestellt werden. Diese Konstruktion sorgt dafür, dass PowerShell zuerst den Ausdruck in den Klammern auswertet und dann wie eine Variable innerhalb des Texts auflöst.

Das Ergebnis weist hierbei möglicherweise sehr viele (und unerwünschte) Nachkommastellen auf. Mehr Kontrolle über die Ausgabe erhalten Sie mit dem Formatierungsoperator *-f*. Er erwartet links eine Textschablone und rechts die Informationen, die in diese Textschablone eingesetzt werden sollen. Innerhalb der Textschablone findet sich in geschweiften Klammern ein Platzhalter, der festlegt, wo genau die Informationen in den Text einzufügen sind. Der Platzhalter besteht aus einer fortlaufenden Indexzahl (also *0* für den ersten Parameter, *1* für den zweiten und so fort) und danach optional einem Doppelpunkt und der gewünschten Formatierung. Der Platzhalter *{0:0}* steht also für den ersten Platzhalter, der Zahlen ohne Nachkommastellen anzeigen und dabei runden soll. Der Platzhalter *{0:0.00}* würde dagegen eine Zahl mit genau zwei Nachkommastellen erzeugen und *{0:0000}* stünde für stets mindestens vierstellige Zahlen. Mehr zum Formatierungsoperator und seinen sonstigen Möglichkeiten erfahren Sie etwas später in diesem Kapitel.

Text über ein Dialogfeld erfragen

Sie möchten Text über ein separates Dialogfeld vom Benutzer erfragen.

Lösung

Greifen Sie auf die .NET-Bibliothek *Microsoft.VisualBasic.Interaction* zu und verwenden Sie daraus die Methode *InputBox()*:

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic') →  
| Out-Null  
PS > $ergebnis = [Microsoft.VisualBasic.Interaction]::InputBox →  
('Geben Sie einen Namen ein!', 'Name erforderlich', 'Vorgabe')  
PS > $ergebnis  
Eingabe
```

Hintergrund

Bevor Sie die Methoden aus der .NET-Bibliothek *Microsoft.VisualBasic.Interaction* nutzen können, muss diese Bibliothek zuerst geladen werden, denn sie gehört nicht zu den Standardbibliotheken, die PowerShell automatisch lädt. Dies braucht pro PowerShell-Sitzung nur einmal zu geschehen.

Danach rufen Sie die statische Methode *InputBox()* aus dem Typ *Microsoft.VisualBasic.Interaction* auf. Sie verhält sich wie die gleichnamige Funktion aus VisualBasic und erwartet drei Argumente: die Eingabeaufforderung, den Dialogfeldtitel und eine Vorgabe, die im Textfeld vorgeschlagen wird.

Eine andere Methode aus demselben Typ heißt *MsgBox()* und kann Ergebnisse als Dialogfeld anzeigen:

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic') →
| Out-Null
PS > $geburtstag = [Microsoft.VisualBasic.Interaction]::InputBox →
('Ihr Geburtstag?', 'Geburtsdatum eingeben', '1.1.1980')
PS > $differenz = New-TimeSpan $geburtstag
PS > [Microsoft.VisualBasic.Interaction]::MsgBox( ('Sie sind {0:0,0} Tage alt.' →
-f $differenz.Days), 64, 'Alter in Tagen')
Ok
```

New-Timespan berechnet eine Zeitdifferenz zwischen dem angegebenen Datum und heute. Dabei akzeptiert *New-Timespan* Datumsangaben im regionalen Datumsformat. *MsgBox()* verlangt in diesem Beispiel drei Argumente: den Ausgabertext, eine Kennziffer für die einzublendenden Symbole und Schaltflächen sowie den Text für die Titelleiste des Dialogfelds.

Text aus einer Datei lesen

Sie möchten Text aus einer Textdatei lesen und diesen Text anschließend weiterverarbeiten.

Lösung

Verwenden Sie *Get-Content* und speichern Sie das Ergebnis in einer Variablen. *Get-Content* liefert den Text aus der Datei zeilenweise als Feld (Array), sodass Sie die einzelnen Textzeilen innerhalb einer Schleife bearbeiten können:

```
PS > $text = Get-Content $env:windir\windowsupdate.log -ReadCount 0
PS > Foreach ($zeile in $text) { "Gelesene Zeile: '$zeile'" }
Gelesene Zeile: '2008-02-13    07:35:31:129    1128    1394    DnldMgr ***** ...
Gelesene Zeile: '2008-02-13    07:35:31:129    1128    1394    DnldMgr * Regulation call
...
```


Möchten Sie den Inhalt der Textdatei lieber nicht zeilenweise lesen, sondern als einen Gesamttext, leiten Sie das Ergebnis weiter an *Out-String*:

```
PS > $text = Get-Content $env:windir\windowsupdate.log -ReadCount 0
PS > $text.GetType().Name
Object[]
PS > $text = Get-Content $env:windir\windowsupdate.log -ReadCount 0 | Out-String
PS > $text.GetType().Name
String
PS > $text.Length
1951381
PS > $text.Substring(1,300)
011-02-13  2  07:35:31:129    1128    1394    DnldMgr ***** DnldMgr: Regulation
Refresh ...
2011-02-13    07:35:31:129    1128    1394    DnldMgr  * Regulation call complete.
0x00000000 ...
2011-02-13    07:35:31:129    1128    1394    DnldMgr ***** DnldMgr: New download
...
```

Hintergrund

Get-Content liest den Inhalt einer Textdatei zeilenweise und übergibt jede gelesene Zeile normalerweise in Echtzeit an die interne PowerShell-Pipeline. Dies ist sehr zeitaufwändig, weswegen *Get-Content* oft als langsam empfunden wird. Gibt man dagegen für den Parameter *-ReadCount* den Wert *0* an, wird die Textdatei zwar immer noch zeilenweise ausgelesen, aber das Ergebnis erst weitergegeben, wenn die gesamte Datei gelesen ist. Dies beschleunigt den Lesevorgang erheblich:

```
PS > Measure-Command { Get-Content $env:windir\windowsupdate.log }
Days          : 0
Hours         : 0
(...)
TotalMinutes  : 0,005751065
TotalSeconds  : 0,3450639
TotalMilliseconds : 345,0639

PS > Measure-Command { Get-Content $env:windir\windowsupdate.log -ReadCount 0 }
Days          : 0
Hours         : 0
(...)
TotalMinutes  : 0,0003307166666666667
TotalSeconds  : 0,019843
TotalMilliseconds : 19,843
```

Dieser Zeitgewinn wird wieder zunichte gemacht, wenn Sie das Ergebnis von *Get-Content* anschließend in der Pipeline weiterverarbeiten. Sie können den Zeitgewinn also nur erreichen, wenn Sie die gelesenen Dateiinhalte anschließend auch ohne Einsatz der PowerShell-Pipeline auswerten. Dazu zwei Beispiele.

Der folgende Code filtert aus der *windowsupdate.log*-Datei alle Zeilen aus, in denen die Wortfolge »successfully installed« vorkommt, und setzt dazu die Pipeline ein:

```
PS > Measure-Command { Get-Content $env:windir\windowsupdate.log -Encoding UTF8 | →
    Where-Object { $_ -like '*successfully installed*' } }
Days           : 0
Hours          : 0
(...)
TotalSeconds   : 1,7183189
TotalMilliseconds : 1718,3189
```

Ein mehr als doppelt so schneller Ansatz verzichtet auf die Pipeline und verwendet stattdessen die *Foreach*-Schleife:

```
PS > Measure-Command { Foreach ($line in (Get-Content →
    $env:windir\windowsupdate.log -Encoding UTF8)) { if ($line -like →
    '*successfully installed*') { $line } } }
Days           : 0
Hours          : 0
(...)
TotalSeconds   : 0,7154899
TotalMilliseconds : 715,4899
```

Get-Content liest den Inhalt einer Textdatei und verwendet dabei als Standardbegrenzerzeichen den Zeilenumbruch. Nach jeder Zeile wird der Lesevorgang also unterbrochen und das Ergebnis zeilenweise in ein Feld geschrieben. So können Sie das Ergebnis zeilenweise ansprechen:

```
PS > $text = Get-Content $env:windir\windowsupdate.log -ReadCount 0
PS > $text[0]
2011-06-01      07:35:31:129      1128      1394      DnldMgr ***** DnldMgr: Regulation
Refresh ...
PS > $text[2]
2011-06-01      07:35:31:129      1128      1394      DnldMgr ***** DnldMgr: New download
job ...
PS > $text.Count
17129
PS > "Die Datei enthält $($text.Count) Zeilen."
Die Datei enthält 17129 Zeilen.
```

Geben Sie mit dem Parameter *-delimiter* ein eigenes Trennzeichen an, trennt *Get-Content* den Text an diesem Trennzeichen und nicht mehr am Zeilenumbruch. Für tabulator-separierte Textdateien könnten Sie also folgendermaßen vorgehen und nun die einzelnen Felder ansprechen:

```
PS > $text = Get-Content $env:windir\windowsupdate.log -delimiter "`t" -ReadCount 0
PS > $text[0]
2011-06-01
PS > $text[1]
07:35:31:129
PS > $text[2]
1128
PS > $text[3]
1394
PS > $text[4]
DnldMgr
PS > $text[5]
***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-4A52D1EFE18D}]
*****2008-02-13
```

ACHTUNG Weil in diesem Beispiel die Textinformationen nun am Tabulatorzeichen getrennt werden und nicht mehr am Zeilenumbruch, verschmelzen jeweils das letzte tabulatorseparierte Element einer Zeile und das erste der folgenden Zeile zu einem Text. Leider ist es nicht möglich, mehrere Trennzeichen anzugeben.

Wollen Sie den Text überhaupt nicht aufteilen und als Array einlesen, leiten Sie das Ergebnis von *Get-Content* weiter an *Out-String*. Beschleunigen Sie in diesem Fall den Lesevorgang unbedingt mit der Option *-ReadCount 0*:

```
PS > $a = Get-Content $env:windir\windowsupdate.log -ReadCount 0 | Out-String
```

Out-String ist notwendig, weil *Get-Content* über keine eigene Möglichkeit verfügt, Texte als einen Gesamttext zu lesen. Der Umweg über *Out-String* ist in vielen Fällen eine gangbare Alternative, kann den Text jedoch möglicherweise verändern: Da *Get-Content* den Originaltext an Zeilenumbrüchen auftrennt und *Out-String* die Zeilen anschließend wieder zusammenfügt, werden die Zeilenumbruch-Zeichen von *Out-String* bestimmt und entsprechen möglicherweise nicht mehr den Zeilenumbruch-Zeichen des Originaltexts.

Darüber hinaus ist dieser Ansatz relativ langsam. Schnellere Ergebnisse ohne Veränderung des Originaltexts erhalten Sie mit den Low-Level-Funktionen von .NET Framework:

```
PS > [System.IO.File]::ReadAllText("c:\sometextfile.txt")
```

Allerdings kann man auf diese Weise nur Dateien lesen, die nicht im Gebrauch anderer Programme sind, weil die Dateien zum exklusiven Lesen geöffnet werden. Systemlogbücher lassen sich damit also häufig nicht lesen. Möchte man Dateien zum nicht-exklusiven Lesen öffnen, sind wenige zusätzliche Zeilen notwendig:

```
PS > $file = [System.IO.File]::Open("$env:windir\windowsupdate.log", 'Open', -->
    'Read', 'ReadWrite')
PS > $reader = New-Object System.IO.StreamReader($file)
PS > $text = $reader.ReadToEnd()
PS > $reader.Close()
PS > $file.Close()
```

Zeilenumbruch oder Anführungszeichen in Texten

Sie möchten in einem Text einen Zeilenumbruch einfügen oder fragen sich, wie man innerhalb eines Texts Anführungszeichen angeben kann, ohne dadurch den Text zu beenden.

Lösung

Begrenzen Sie Ihren Text mit doppelten Anführungszeichen und verwenden Sie das besondere Backtick-Zeichen zusammen mit einem der Sonderzeichen aus Tabelle 1.1 (siehe Seite 29). Einen mehrzeiligen Text gestalten Sie zum Beispiel so:

```
PS > "Dieser Text besteht aus zwei Zeilen`nHier ist die zweite Zeile"
Dieser Text besteht aus zwei Zeilen
Hier ist die zweite Zeile
```

Alle nicht in der Tabelle aufgeführten Zeichen werden wörtlich ausgegeben, wenn sie direkt hinter dem Backtick folgen. So fügen Sie beispielsweise Anführungszeichen in einen Text ein:

```
PS > "Dieser Text enthält `\"Anführungszeichen`\"!"
Dieser Text enthält "Anführungszeichen"!
```



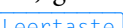
Anführungszeichen lassen sich auch in einen Text einfügen, indem Sie sie doppelt hintereinander schreiben:

```
PS > "Dieser Text enthält \"\"Anführungszeichen\"\"!"
Dieser Text enthält "Anführungszeichen"!
PS > 'Dieser Text enthält \'\'Anführungszeichen\'\'!'
Dieser Text enthält 'Anführungszeichen'!
```

Haben Sie als Textbegrenzer ein anderes Anführungszeichen gewählt als das, was Sie einfügen wollen, besteht ohnehin kein Problem:

```
PS > "Dieser Text enthält 'Anführungszeichen'!"
Dieser Text enthält 'Anführungszeichen'!
PS > 'Dieser Text enthält "Anführungszeichen"!
Dieser Text enthält "Anführungszeichen"!
```

Hintergrund

Das besondere Backtick-Zeichen erreichen Sie über die deutsche Tastaturbelegung auf etwas umständlichem Weg: Halten Sie die -Taste gedrückt, geben Sie ein Apostroph ein (Taste rechts neben der Taste ) und drücken Sie dann . Achten Sie darauf, dass das Backtick-Zeichen seine besondere Funktion nur erfüllt, wenn der Text in doppelten Anführungszeichen eingeschlossen ist. Text in einfachen Anführungszeichen würde das Backtick-Zeichen wörtlich ausgeben.

Die folgenden Sonderzeichen stehen innerhalb doppelter Anführungszeichen zur Verfügung:

Sonderzeichen	Beschreibung
`0	Null-Zeichen
`a	Alarm-Zeichen. Erzeugt einen Piepton. "Eine Warnung mit Piepton!`a"
`b	Rück-Zeichen. Versetzt die Cursorposition um ein Zeichen nach links und überschreibt so das vorherige Zeichen, wenn der Text in der Konsole ausgegeben wird. "Guten Tag`b`b`bMorgen"
`f	Seitenumbruch. Erzeugt beim Ausdruck einen Seitenvorschub. Der folgende Text wird auf einer neuen Seite ausgegeben.
`n	Zeilenumbruch. Der folgende Text wird in einer neuen Zeile ausgegeben. "Erste Zeile`nZweite Zeile"
`r	Wagenrücklauf. Setzt den Cursor an den Beginn der aktuellen Zeile.
`t	Tabulatorzeichen
`v	Vertikaler Tabulator
`(Zeichen)	Das hinter dem Backtick angegebene Zeichen wird wörtlich und ohne Umwandlung ausgegeben Beispiele: `` oder `\$variable
"	Ein einfaches Anführungszeichen in Texten, die mit einfachen Anführungszeichen begrenzt sind
""	Ein doppeltes Anführungszeichen in Texten, die mit doppelten Anführungszeichen begrenzt sind

Tabelle 1.1 Sonderzeichen in Texten

TIPP

Here-Strings unterscheiden ebenfalls zwischen einfachen und doppelten Anführungszeichen. Verwenden Sie für *Here-Strings* einfache Anführungszeichen, werden keine Ersetzungen im Text durchgeführt.

Informationen in einen Text einfügen

Sie wollen dynamische Informationen in einen statischen Text einfügen, zum Beispiel einen Pfadnamen mit einer Beschreibung versehen.

Lösung

Sofern die dynamische Information in einer Variablen vorliegt, setzen Sie den Text in doppelte Anführungszeichen und geben die Variable im Text direkt an. Die Variable wird automatisch von PowerShell durch ihren Inhalt ersetzt:

```
PS > "Der Windows-Ordner liegt hier: $env:windir"  
Der Windows-Ordner liegt hier: C:\Windows
```

Alternativ können Sie die Informationen auch mit dem »+«-Operator verketteten, wenn die Informationen aus Text bestehen oder in Text umgewandelt werden können:

```
PS > 'Der Windows-Ordner liegt hier: ' + $env:windir  
Der Windows-Ordner liegt hier: C:\Windows
```

Muss die Information erst errechnet werden, verwenden Sie Unterausdrücke, die in runden Klammern stehen:

```
PS > 'Das Ergebnis lautet: ' + (4+7)  
Das Ergebnis lautet: 11
```

Unterausdrücke können auch als Variablen direkt in den Text integriert werden. Dazu wird vor die runden Klammern ein `$`-Zeichen gesetzt:

```
PS > "Das Ergebnis lautet: $(4+7)"  
Das Ergebnis lautet: 11
```

Stammt die Information aus einer Eigenschaft eines Objekts, müssen Unterausdrücke verwendet werden, weil die automatische Umwandlung sich stets nur auf die Variable selbst bezieht:

```
PS > "Die aktuelle Konsolen-Hintergrundfarbe: $($host.UI.RawUI.BackgroundColor)"  
Die aktuelle Konsolen-Hintergrundfarbe: Magenta
```

Wollen Sie Text als Schablone verwenden, setzen Sie den Formatierungsoperator `-f` ein und geben im Schablonentext Platzhalter für die Informationen an. Diese Platzhalter bestehen aus geschweiften Klammern und einer Indexzahl. Hinter dem Formatierungsoperator geben Sie die Werte an, die anstelle der Platzhalter eingefügt werden sollen:

```
PS > 'Windows-Ordner: {0}, Hintergrundfarbe: {1}' -f $env:windir, →  
($host.UI.RawUI.BackgroundColor)  
Windows-Ordner: C:\Windows, Hintergrundfarbe: Magenta
```

Hintergrund

Die Verknüpfung mehrerer Informationen mit dem »+«-Operator ist zwar möglich, aber unübersichtlich und fehlerträchtig. Sie setzt nämlich voraus, dass die Informationen, die Sie verknüpfen, vom selben Datentyp sind. PowerShell richtet sich dabei stets nach dem Datentyp des ersten Elements. Ist das erste Element ein Text, werden alle weiteren Elemente ebenfalls in Text verwandelt. Ist das nicht möglich, kommt es zu einem Fehler:

```
PS > 'Ergebnis: ' + 5+6
Ergebnis: 56
PS > 'Ergebnis: ' + Get-Date
Sie müssen auf der rechten Seite des Operators "+" einen Wertausdruck angeben.
Bei Zeile:1 Zeichen:15
+ 'Ergebnis: ' + <<<< Get-Date
```

Dieses Problem kann umgangen werden, indem Sie Rechenoperationen oder den Zugriff auf Objekteigenschaften als separate Unterausdrücke in runden Klammern zusammenfassen:

```
PS > 'Ergebnis: ' + (5+6)
Ergebnis: 11
PS > 'Ergebnis: ' + (Get-Date)
Ergebnis: 04/10/2008 13:54:44
```

Hierbei wertet PowerShell den Ausdruck in den runden Klammern zuerst aus und ersetzt ihn anschließend durch das Ergebnis der Auswertung.

Da sich PowerShell bei der automatischen Umwandlung verschiedener Datentypen stets nach dem ersten Ausdruck richtet, muss der erste Ausdruck ein Text sein. Andernfalls führt die Umwandlung zu einem falschen Datentyp:

```
PS > Get-Service | ForEach-Object { $_.CanShutdown + " " + $_.Name }
Der Vorgang "[System.Boolean] + [System.String]" ist nicht definiert.
Bei Zeile:1 Zeichen:48
+ Get-Service | ForEach-Object { $_.CanShutdown + <<<< " " + $_.Name }
Der Vorgang "[System.Boolean] + [System.String]" ist nicht definiert.
Bei Zeile:1 Zeichen:48
+ Get-Service | ForEach-Object { $_.CanShutdown + <<<< " " + $_.Name }
(...)
```

Sie müssten hier also den ersten Wert zuerst in einen Text konvertieren, beispielsweise indem Sie *ToString()* aufrufen:

```
PS > Get-Service | ForEach-Object { $_.CanShutdown.ToString() + " " + $_.Name }
False AEADIFilters
False AeLookupSvc
False ALG
False Appinfo
False AppMgmt
(...)
```

Übersichtlicher ist es, die dynamischen Informationen als Variable direkt in den Text zu integrieren. Sofern Sie doppelte Anführungszeichen als Textbegrenzer verwenden, werden die Variablen automatisch durch ihren jeweiligen Inhalt ersetzt:

```
PS > "Benutzer: $env:username"
Benutzer: Tobias
```

Eine besondere Variablenform sind Direktvariablen. Sie werden wie normale Variablen mit dem »\$«-Zeichen eingeleitet. Danach folgen runde Klammern und der Inhalt der Direktvariable wird vom Ausdruck innerhalb der runden Klammern errechnet.

Damit lassen sich zum Beispiel Unterstreichungen mit automatisch berechneter Länge einfügen. Im folgenden Beispiel wird zuerst der Benutzername und dann ein Zeilenumbruch ausgegeben. Dahinter folgt eine Direktvariable, die genauso viele Gleichheitszeichen ausgibt, wie der Benutzername lang ist:

```
PS > "$env:username`n$('=' * ($env:username).length)"
Tobias
=====
```

Die Vorgänge hierbei lassen sich besser verstehen, wenn man den Code auf mehrere Zeilen aufteilt:

```
PS > $username = $env:username
PS > $username
Tobias
PS > '=' * $username.length
=====
```

HINWEIS

Der Multiplikator-Operator »*« erfüllt bei Texten eine besondere Bedeutung: Er wiederholt den Text links von ihm so oft, wie rechts von ihm angegeben ist:

```
PS > 'Hallo!' * 5
Hallo!Hallo!Hallo!Hallo!Hallo!
```

Wollen Sie mehrere dynamische Informationen in einen festen Text integrieren oder die dynamischen Informationen besonders formatieren, zum Beispiel mit einer festgelegten Anzahl von Nachkommastellen, eignet sich der Operator `-f` am besten. Er trennt den statischen Text von den dynamischen Informationen, indem Sie im statischen Text an den gewünschten Stellen Platzhalter einfügen. Die dynamischen Informationen, die anstelle der Platzhalter eingefügt werden, geben Sie hinter dem Operator als kommaseparierte Liste (genau genommen also als Feld) an:

```
PS > 'Ihr Name lautet {0} und Ihr Windows lebt hier: {1}' -f $env:username, $env:windir
Ihr Name lautet Tobias und Ihr Windows lebt hier: C:\Windows
```

Der Vorteil dieser Schreibweise ist unter anderem, dass der statische Text besser lokalisierbar ist, also leicht in unterschiedliche Sprachen übersetzt werden kann. Denken Sie aber daran, dass hinter dem

Operator die tatsächlichen Werte folgen müssen, die für die Platzhalter eingesetzt werden sollen. Müssen diese Werte zuerst errechnet werden oder verwenden Sie Objekteigenschaften, setzen Sie die einzelnen Werte in runde Klammern, damit sie als Unterausdruck zuerst ausgewertet werden:

```
PS > # Vergleichen Sie die Unterschiede:
PS > 'Das Ergebnis lautet {0}' -f 4+10
Das Ergebnis lautet 410
PS > 'Das Ergebnis lautet {0}' -f (4+10)
Das Ergebnis lautet 14
```

Dynamische Informationen formatieren

Sie möchten Informationen in besonderen Formaten in Text verwandeln, also zum Beispiel bestimmen, wie viele Nachkommastellen eine Zahl verwendet, oder Textspalten links- oder rechtsbündig ausgeben.

Lösung

Verwenden Sie den Formatierungsoperator `-f` und geben Sie für jeden Platzhalter das gewünschte Format an. Die folgende Zeile gibt den freien Speicherplatz eines Laufwerks in Megabyte mit genau einer Nachkommastelle an (siehe auch Tabelle 1.4 auf Seite 38):

```
PS > $freespace = (Get-WmiObject Win32_LogicalDisk -filter 'Name="C:"').FreeSpace
PS > 'Freier Speicherplatz auf C: {0:0.0} MB' -f ($freespace/1MB)
Freier Speicherplatz auf C: 9407,0 MB
```

Diese Zeile gibt den aktuellen Wochentag aus (siehe auch Tabelle 1.3 auf Seite 37):

```
PS > 'Wochentag: {0:dddd}' -f (Get-Date)
Wochentag: Dienstag
```

Den Monat erhalten Sie über diese Zeile:

```
PS > 'Monat: {0:MMMM}' -f (Get-Date)
Monat: April
```

Die nächsten Zeilen erzeugen eine unterstrichene Spaltenüberschrift, wobei die Unterstreichung genauso lang ist wie der Überschriftentext:

```
PS > $text = 'Inventarisierungsreport September'
PS > $überschrift = '{0}{1}{2}' -f $text, "`n", ('=' * $text.Length)
PS > $überschrift
Inventarisierungsreport September
=====
```

Die Formatierungsinformation darf auch in einer Variablen gespeichert werden, wenn Sie mehr als eine Zeile damit formatieren möchten. Das folgende Beispiel erzeugt aus einer Liste mit Wertpaaren eine zweispaltige Ausgabe. Die erste Spalte ist formatiert als sechststellige Zahl. Die zweite Spalte ist rechtsbündig und neun Zeichen breit sowie als Währung formatiert:

```
PS > $format = '{0:D6} {1,9:C}'
PS > (4, 33.20), (12, 8.34), (2, 44.30) | ForEach-Object {'Anzahl    Preis'} →
    { $format -f $_[0], $_[1] }
Anzahl    Preis
000004    33,20 ?
000012     8,34 ?
000002    44,30 ?
```

Und wenn Sie für Logbuchdateien einen Namen basierend auf einem Zeitstempel benötigen, können Sie den Dateinamen zum Beispiel vom aktuellen Datum ableiten:

```
PS > $dateiname = 'mylog{0:dd-MM-yyyy HH-mm-ss}.log' -f (Get-Date)
PS > $dateiname
mylog13-02-2011 12-17-38.log
```

Hintergrund

Der Formatierungsoperator `-f` erwartet auf der linken Seite die Textschablone und auf der rechten Seite die Werte, die in die Schablone eingefügt werden sollen. Innerhalb der Textschablone markieren geschweifte Klammern mit einer Indexzahl die Platzhalter, in die die Informationen auf der rechten Seite eingefügt werden.

Verwenden Sie als Platzhalter lediglich geschweifte Klammern mit der Indexzahl, werden die dynamischen Informationen anstelle der Platzhalter in den Text eingefügt und mit dem Standardformat angezeigt.

```
PS > 'Eingefügter Wert: {0}.' -f 1000
Eingefügter Wert: 1000.
```

Mehr Kontrolle erhalten Sie, wenn Sie im jeweiligen Platzhalter genau festlegen, wie die Information dargestellt werden soll. Wie ein Platzhalter den eingefügten Wert formatieren soll, bestimmen Sie mit einem Doppelpunkt und/oder einem Komma:

- **Doppelpunkt** Hinter dem Doppelpunkt folgt eine Beschreibung des Formats. Damit bestimmen Sie zum Beispiel, wie viele Nachkommastellen eine Zahl haben soll oder welche Informationen aus einer Datumsangabe eingefügt werden sollen.
- **Komma** Hinter dem Komma geben Sie an, wie breit dieses Feld sein soll und ob es links- oder rechtsbündig ist

Möchten Sie eine Zahl zum Beispiel mit zwei Nachkommastellen ausgeben, gehen Sie so vor:

```
PS > 'Eingefügter Wert: {0:0.00}.' -f 1000
Eingefügter Wert: 1000,00.
```

Möchten Sie den Wert in ein Feld mit 30 Zeichen Breite linksbündig ausgeben, gehen Sie so vor:

```
PS > 'Eingefügter Wert: {0,30}.' -f 1000
Eingefügter Wert:                1000.
```

Rechtsbündigkeit erreichen Sie durch negative Zahlen:

```
PS > 'Eingefügter Wert: {0,-30}.' -f 1000
Eingefügter Wert: 1000                .
```

Wollen Sie beide Formatierungen kombinieren, geben Sie zuerst das Komma und dann den Doppelpunkt an. Eine Zahl mit zwei Nachkommastellen in einem 30 Zeichen breiten linksbündigen Feld erhalten Sie also folgendermaßen:

```
PS > 'Eingefügter Wert: {0,30:0.00}' -f 1000
Eingefügter Wert:                1000,00
```

Die Feldbreite eignet sich ideal dazu, Informationen bündig spaltenweise auszugeben:

```
PS > dir $env:windir\web\wallpaper -filter *.jpg -recurse | ForEach-Object -->
    { '{0,-40} = {1,10} Bytes' -f $_.Name, $_.Length }
img13.jpg                = 1231580 Bytes
img14.jpg                = 1526083 Bytes
img15.jpg                = 1492201 Bytes
img16.jpg                = 1236846 Bytes
img17.jpg                = 1388763 Bytes
img18.jpg                = 691271 Bytes
img19.jpg                = 1472847 Bytes
img20.jpg                = 1226804 Bytes
(...)
```

Durch die Kombination von Feldbreite und Format lassen sich in wenigen Zeilen übersichtlich formatierte Reports generieren. Im folgenden Beispiel wird im zweiten Platzhalter das Datum mit Wochentag ausgegeben:

```
PS > dir $env:windir\web\wallpaper -filter *.jpg -recurse | ForEach-Object -->
    { '{0,-40} erzeugt: {1,30:D}' -f $_.Name, $_.CreationTime }
img13.jpg                erzeugt:    Montag, 13. Juli 2010
img14.jpg                erzeugt:    Montag, 13. Juli 2010
img15.jpg                erzeugt:    Montag, 13. Juli 2010
img16.jpg                erzeugt:    Montag, 13. Juli 2010
img17.jpg                erzeugt:    Montag, 13. Juli 2010
img18.jpg                erzeugt:    Montag, 13. Juli 2010
img19.jpg                erzeugt:    Montag, 13. Juli 2010
img20.jpg                erzeugt:    Montag, 13. Juli 2010
(...)
```

Der zweite Platzhalter ist mindestens 30 Zeichen breit und verwendet das lange Datumsformat (Kennung »D«). Die verschiedenen Datumsformate geben die folgenden beiden Zeilen als Übersicht aus:

```
PS > $datum = Get-Date
PS > "d","D","f","F","g","G","m","r","s","t","T","u","U","y","dddd", →
    "MMM dd yyyy","M/yy","dd-MM-yy" | ForEach-Object {"DATUM mit $_ : {0}" →
    -f $datum.ToString($_)}
```

DATUM mit d : 14.02.2011
 DATUM mit D : Montag, 14. Februar 2011
 DATUM mit f : Montag, 14. Februar 2011 09:14
 DATUM mit F : Montag, 14. Februar 2011 09:14:38
 DATUM mit g : 14.02.2011 09:14
 DATUM mit G : 14.02.2011 09:14:38
 DATUM mit m : 14 Februar
 DATUM mit r : Mon, 14 Feb 2011 09:14:38 GMT
 DATUM mit s : 2011-02-14T09:14:38
 DATUM mit t : 09:14
 DATUM mit T : 09:14:38
 DATUM mit u : 2011-02-14 09:14:38Z
 DATUM mit U : Montag, 14. Februar 2011 08:14:38
 DATUM mit y : Februar 2011
 DATUM mit dddd : Montag
 DATUM mit MMM dd yyyy : Februar 14 2011
 DATUM mit M/yy : 2.11
 DATUM mit dd-MM-yy : 14-02-11

Die folgenden Tabellen fassen die Formatierungsoptionen für Datum und Zeit zusammen. Beachten Sie, dass bei den Formatierungsoptionen die Groß- und Kleinschreibung wichtig ist. Die folgende Tabelle 1.2 listet die Formatierungsoptionen für komplette Datums- und Zeitangaben auf.

Symbol	Typ	Aufruf	Ergebnis
d	Kurzes Datumsformat	"{0:d}" -f \$wert	01.06.2011
D	Langes Datumsformat	"{0:D}" -f \$wert	Mittwoch, 1. Juni 2011
t	Kurzes Zeitformat	"{0:t}" -f \$wert	10:53
T	Langes Zeitformat	"{0:T}" -f \$wert	10:53:56
f	Datum & Uhrzeit komplett (kurz)	"{0:f}" -f \$wert	Mittwoch, 1. Juni 2011 10:53
F	Datum & Uhrzeit komplett (lang)	"{0:F}" -f \$wert	Mittwoch, 1. Juni 2011 10:53:56
g	Standard-Datum (kurz)	"{0:g}" -f \$wert	01.06.2011 10:53
G	Standard-Datum (lang)	"{0:G}" -f \$wert	01.06.2011 10:53:56
M	Tag des Monats	"{0:M}" -f \$wert	01 Juni
r	RFC1123-Datumsformat	"{0:r}" -f \$wert	Mi, 01 Jun 2011 10:53:56 GMT

Tabelle 1.2 Datumswerte formatieren

Symbol	Typ	Aufruf	Ergebnis
s	Sortierbares Datumsformat	"{0:s}" –f \$wert	2011-06-01T10:53:56
u	Universell sortierbares Datumsformat	"{0:u}" –f \$wert	2011-06-01 10:53:56Z
U	Universell sortierbares GMT-Datumsformat	"{0:U}" –f \$wert	Mittwoch, 1. Juni 2011 08:53:56
Y	Jahr/Monats-Muster	"{0:Y}" –f \$wert	Juni 2011

Tabelle 1.2 Datumswerte formatieren (*Fortsetzung*)

Die folgende Tabelle 1.3 fasst die Formatierungsoptionen für individuelle Datums- und Zeitbestandteile zusammen. Mit diesen Formatierungsoptionen können Sie aus Datums- und Zeitangaben individuelle Informationen extrahieren. Die Formatierungsoptionen lassen sich auch kombinieren, um so eigene Datumsformate zusammenzustellen.

Symbol	Typ	Aufruf	Ergebnis
dd	Tag	"{0:dd}" –f \$wert	01
ddd	Tagname (Kürzel)	"{0:ddd}" –f \$wert	Mi
dddd	Tagname (ausgeschrieben)	"{0:dddd}" –f \$wert	Mittwoch
gg	Ära	"{0:gg}" –f \$wert	n. Chr.
hh	Stunde zweistellig	"{0:hh}" –f \$wert	10
HH	Stunde zweistellig (24-Stunden)	"{0:HH}" –f \$wert	10
mm	Minute	"{0:mm}" –f \$wert	53
MM	Monat	"{0:MM}" –f \$wert	09
MMM	Monatsname (Kürzel)	"{0:MMM}" –f \$wert	Jun
MMMM	Monatsname (ausgeschrieben)	"{0:MMMM}" –f \$wert	Juni
ss	Sekunde	"{0:ss}" –f \$wert	56
tt	AM oder PM (nur englisch)	"{0:tt}" –f \$wert	
yy	Jahr zweistellig	"{0:yy}" –f \$wert	11
yyyy	Jahr vierstellig	"{0:YY}" –f \$wert	2011
zz	Zeitzone (kurz)	"{0:zz}" –f \$wert	+02
zzz	Zeitzone (lang)	"{0:zzz}" –f \$wert	+02:00

Tabelle 1.3 Datumswerte individuell formatieren

Der Formatierungsoperator kann auch dafür eingesetzt werden, Zahlenwerte hexadezimal auszugeben. Verwenden Sie dazu entweder die Kennung »x« (hexadezimale Werte mit Kleinbuchstaben) oder »X« (hexadezimale Werte mit Großbuchstaben):

```
PS > '{0:x}' -f 45688
b278
```

Führende Nullen erhalten Sie, indem Sie hinter der Kennung die Mindestanzahl der Stellen angeben. Die folgende Zeile liefert einen hexadezimalen Wert in Großbuchstaben mit mindestens acht Stellen:

```
PS > '{0:X8}' -f 45688
0000B278
```

In der folgenden Tabelle 1.4 sind die Formatierungsoptionen für die Formatierung von Zahlen und Zahlenformaten aufgelistet.

Symbol	Typ	Aufruf	Ergebnis
#	Zahl-Platzhalter	"{0:(#).##}" -f \$wert	(1000000)
%	Prozentwert	"{0:0%}" -f \$wert	100000000%
,	Tausender Trennzeichen	"{0:0,0}" -f \$wert	1.000.000
..	Ganzzahliges Vielfaches von 1.000	"{0:0,.}" -f \$wert	1000
.	Dezimalpunkt	"{0:0.0}" -f \$wert	1000000,0
0	0-Platzhalter	"{0:00.0000}" -f \$wert	1000000,0000
c	Währung (currency)	"{0:c}" -f \$wert	1.000.000,00 _
d	Dezimalzahl (decimal)	"{0:d}" -f \$wert	1000000
e	Wissenschaftlich (scientific)	"{0:e}" -f \$wert	1,000000e+006
e	Exponentenplatzhalter	"{0:00e+0}" -f \$wert	10e+5
f	Festkommazahl (fixed point)	"{0:f}" -f \$wert	1000000,00
g	Generisch (general)	"{0:g}" -f \$wert	1000000
n	Tausendertrennzeichen	"{0:n}" -f \$wert	1.000.000,00
x	Hexadezimal	"0x{0:x4}" -f \$wert	0x4240

Tabelle 1.4 Zahlen formatieren

Die Formatierung wird grundsätzlich von allen Datentypen unterstützt, die über die Methode `toString()` verfügen:

```
PS > [appdomain]::currentdomain.getassemblies() | ForEach-Object →
    {$_ .GetExportedTypes() | ? {! $_.IsSubclassof([System.Enum])}} →
    | ForEach-Object { $Methods = $_.getmethods() | ? {$_ .name -eq "tostring"} →
    | ForEach-Object{"$ "};
If ($methods -eq "System.String ToString(System.String)") {$_ .fullname}}
System.Enum
System.DateTime
System.DateTimeOffset
System.Byte
System.Convert
System.Decimal
System.Double
System.Guid
System.Int16
System.Int32
System.Int64
System.IntPtr
System.SByte
System.Single
System.UInt16
System.UInt32
System.UInt64
Microsoft.PowerShell.Commands.MatchInfo
```

Zu den unterstützten Datentypen gehört auch *System.Guid*, ein »Globally Unique Identifier«:

```
PS > $guid = [GUID]::NewGUID()
PS > "N","D","B","P" | ForEach-Object {'[GUID]::NewGuid().ToString('{0}') = →
    {1}' -f $_, $GUID.ToString($_)}
[GUID]::NewGuid().ToString('N') = 94a5feaffa0848668f055fea268be867
[GUID]::NewGuid().ToString('D') = 94a5feaf-fa08-4866-8f05-5fea268be867
[GUID]::NewGuid().ToString('B') = {94a5feaf-fa08-4866-8f05-5fea268be867}
[GUID]::NewGuid().ToString('P') = (94a5feaf-fa08-4866-8f05-5fea268be867)
```

Der Operator *-f* ist übrigens eine Abkürzung für die statische .NET-Methode *Format()* des Typs *String*:

```
PS > # Beide Zeilen liefern ein identisches Ergebnis:
PS > '{0:D5}' -f 46
00046
PS > [String]::Format('{0:D5}', 46)
00046
```

Stichwörter in Texten finden

Sie möchten herausfinden, ob eine bestimmte Information in einem Text enthalten ist.

Lösung

Die *String*-Methode *Contains()* findet heraus, ob ein bestimmter Begriff exakt so wie angegeben in einem Text vorhanden ist:

```
PS > "Hallo Welt".Contains("Welt")
True
```

Hierbei wird die Groß- und Kleinschreibung beachtet, der Suchbegriff muss also in genau der angegebenen Schreibweise im Text vorkommen. Analog funktioniert der PowerShell-Operator *-like*, bei dem die Groß- und Kleinschreibung allerdings nicht unterschieden wird:

```
PS > 'Hallo Welt' -like '*welt*'
True
```

Ist Ihnen die Groß- und Kleinschreibung wichtig, verwenden Sie anstelle von *-like* den Operator *-clike*:

```
PS > 'Hallo Welt' -clike '*welt*'
False
```

Die folgende Anweisung überprüft grob, ob es sich bei einem Text um eine E-Mail-Adresse handelt:

```
PS > 'tobias.weltner@powershell.de' -like '*@*.*'
True
```

Entsprechend könnten Sie grob prüfen, ob eine bestimmte Information einer IP-Adresse entspricht:

```
PS > '10.10.255.232' -like '.*.*.*'
True
```

Für eine genauere Prüfung greifen Sie zu regulären Ausdrücken und dem Operator *-match* (siehe »Hintergrund«). Diese Zeile prüft, ob es sich tatsächlich um eine IP-Adresse handelt:

```
PS > $ipadresse = '^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|
[0-9]|[01]?[0-9][0-9]?)$'
PS > '10.10.255.232' -match $ipadresse
True
PS > '10.10.255.280' -match $ipadresse
False
PS > 'hallo' -match $ipadresse
False
```


Wollen Sie prüfen, ob ein Text mit einem bestimmten Wort beginnt oder endet, verwenden Sie *StartsWith()* bzw. *EndsWith()*:

```
PS > 'c:\autoexec.bat'.StartsWith('c:\')
True
PS > 'c:\autoexec.bat'.EndsWith('.bat')
True
```

Hierbei wird Groß- und Kleinschreibung unterschieden. Möchten Sie nicht zwischen Groß- und Kleinschreibung unterscheiden, verwandeln Sie alle Texte vor dem Vergleich mit *ToLower()* in Kleinbuchstaben:

```
PS > 'C:\AUTOEXEC.BAT'.ToLower().StartsWith('c:\')
True
PS > 'C:\AUTOEXEC.BAT'.ToLower().EndsWith('.bat')
True
```

Hintergrund

Um Informationen in Texten zu finden, stehen Ihnen zwei Quellen für Befehle zur Verfügung: die *String*-Methoden von .NET Framework sowie die PowerShell-Operatoren.

Alle PowerShell-Operatoren beginnen mit einem Bindestrich. Es gibt die Operatoren jeweils in drei Fassungen. Die Standardversion unterscheidet nicht zwischen Groß- und Kleinschreibung. Stellen Sie ein »c« vor den Operatornamen, wird Groß- und Kleinschreibung unterschieden. Stellen Sie ein »i« vor, wird Groß- und Kleinschreibung nicht unterschieden. Damit verhält sich die »i«-Variante wie der Standardoperator. Verwenden Sie die »i«-Variante, wenn es Ihnen wichtig ist, zu dokumentieren, dass Sie die Groß- und Kleinschreibung nicht unterscheiden wollen:

```
PS > 'Hallo Welt' -like '*welt*'
True
PS > 'Hallo Welt' -ilike '*welt*'
True
PS > 'Hallo Welt' -clike '*welt*'
False
```

.NET-Befehle werden vom *String*-Objekt bereitgestellt, in dem der Text gespeichert ist. Sie greifen auf diese Befehle über einen Punkt zu, den Sie an den Text anhängen:

```
PS > 'Hallo Welt'.Length
10
```

Suchen Sie nach Textmustern, stehen Ihnen zwei verschiedene Ansätze zur Verfügung. Die einfache Variante verwendet den Operator *-like*, der eine überschaubare Zahl von Platzhalterzeichen unterstützt. Viele einfache Textmuster lassen sich so am einfachsten identifizieren. Tatsächlich wird diese Form der Platzhalter von vielen Dateisystem-Befehlen ebenfalls unterstützt:

Platzhalter	Beschreibung	Beispiel
*	Beliebig viele beliebige Zeichen (oder gar kein Zeichen)	'Hallo Welt' –like '*we*' dir a*
?	Genau ein beliebiges Zeichen	'Hallo Welt' –like 'H?ll*' dir *.t??
[xyz]	Eines der angegebenen Zeichen	'Hallo Welt' –like '[HAK]allo*' dir [abc]*.*
[x–z]	Eines der Zeichen im angegebenen Bereich	dir \$env:windir\system32\[0–9].dll

Tabelle 1.5 Platzhalterzeichen für *–like*

Mehr Genauigkeit bieten reguläre Ausdrücke, die Sie zusammen mit *–match* verwenden. Reguläre Ausdrücke unterstützen viel mehr Platzhalterzeichen, sodass sich Textmuster präziser beschreiben lassen. Allerdings wirken reguläre Ausdrücke deswegen oft komplex, und es kann nicht immer einfach sein, einen regulären Ausdruck selbst zusammenzustellen. Glücklicherweise findet man mit den Internetsuchmaschinen aber meist schnell Beispiele für reguläre Ausdrücke zu den gebräuchlichsten Mustern.

Im einfachsten Fall verwenden Sie für *–match* keine besonderen Platzhalterzeichen. Dann findet *–match* das angegebene Suchwort an beliebiger Stelle im Text. Allerdings dürfen Sie keine der Sonderzeichen verwenden, die die Tabellen unten aufführen, oder müssen diese mit einem »\«-Zeichen jeweils entwerten:

```
PS > "c:\windows\unterordner\datei.txt" -match 'datei'
True
PS > "c:\windows\unterordner\datei.txt" -match '\datei.txt'
False
PS > "c:\windows\unterordner\datei.txt" -match '\\datei.txt'
True
```

Platzhalter	Beschreibung
\d	Eine Zahl
\D	Beliebiges Zeichen außer einer Zahl
\s	Ein Leerzeichen (einschließlich Tabulator und Zeilenumbruch)
\S	Beliebiges Zeichen außer Leerzeichen
\t	Ein Tabulatorzeichen
\w	Ein Buchstabe, eine Ziffer oder ein Unterstrich
\W	Beliebiges Zeichen außer Buchstaben
.	Beliebiges Zeichen

Tabelle 1.6 Platzhalter in regulären Ausdrücken

Platzhalter	Beschreibung
[a–z]	Eines der Zeichen im Bereich
[abc]	Eines der angegebenen Zeichen
\.	Entspricht das Zeichen hinter »\« keinem der Sonderzeichen, wird es wörtlich genommen. In diesem Beispiel wird also ein Punkt erwartet. Ohne »\« würde der Punkt ein Platzhalter für ein beliebiges Zeichen darstellen.

Tabelle 1.6 Platzhalter in regulären Ausdrücken (*Fortsetzung*)

Jeder Platzhalter steht für genau ein Zeichen. Wie oft dieses Zeichen vorkommen darf, bestimmen die sogenannten Quantifizierer, die Sie an den Platzhalter anfügen können.

Quantifizierer	Beschreibung
*	Ausdruck kommt beliebig oft vor (einschließlich keinmal)
?	Ausdruck kommt einmal oder keinmal vor
+	Ausdruck kommt einmal vor
{n}	Ausdruck kommt genau <i>n</i> mal vor
{n,}	Ausdruck kommt mindestens <i>n</i> mal vor
{n,m}	Ausdruck kommt mindestens <i>n</i> mal und höchstens <i>m</i> mal vor

Tabelle 1.7 Quantifizierer in regulären Ausdrücken

Mithilfe von regulären Ausdrücken lässt sich eine IP-Adresse sehr viel präziser beschreiben als mit den wenigen Platzhalterzeichen, die –*like* zur Verfügung stellt. Das folgende Beispiel akzeptiert vier von Punkten getrennte maximal dreistellige Zahlen.

```
PS > '192.168.2.1' -match '\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
True
```

Dieses allgemeine Muster könnte man also auch für Versionsnummern einsetzen. Allerdings sind die Zahlenangaben nicht auf Werte zwischen 0 und 255 begrenzt. Für diese Prüfung müsste der reguläre Ausdruck weiter verfeinert werden. Ein Beispiel haben Sie oben bereits gesehen.

Sie können auch nach verschiedenen ähnlichen Schreibweisen suchen:

```
PS > 'Color' -match 'Colou?r'
True
PS > 'Colour' -match 'Colou?r'
True
```

Mit runden Klammern bilden Sie Unterausdrücke innerhalb der Textmuster. So können Sie prüfen, ob ein weiterer Ausdruck innerhalb des Musters vorkommt, und für den Unterausdruck eigene Quantifizierer angeben. Der nächste Ausdruck erkennt zum Beispiel nicht nur den Monat »November«, sondern auch die abgekürzte Fassung »Nov«:

```
PS > 'November' -match 'Nov(ember)?'
True
PS > 'Nov' -match 'Nov(ember)?'
True
```

Mit dem senkrechten Strich bilden Sie ebenfalls Gruppen. In diesem Fall muss eine der angegebenen Textmuster zutreffen:

```
PS > 'November' -match 'Nov|November'
True
PS > 'Nov' -match 'Nov|November'
True
```

Ohne Anker sucht `-match` das Textmuster an beliebiger Stelle im Text. Mit Ankern bestimmen Sie, wo das Textmuster zu finden sein soll, beispielsweise am Satzanfang oder an einer Wortgrenze.

Anker	Beschreibung
\$	Satzende
Z	Textende
^	Satzanfang
A	Textanfang
b	Wortgrenze
B	Nicht an einer Wortgrenze

Tabelle 1.8 Anker geben an, wo das Textmuster im Text gesucht wird

Der folgende Ausdruck findet das Textmuster an einer beliebigen Stelle im Text:

```
PS > 'Bahnhof' -match 'hof'
True
```

Durch die Angabe eines Ankers sorgen Sie dafür, dass das Suchwort nur gefunden wird, wenn es an einer Wortgrenze beginnt, also nicht Teil eines anderen Worts ist:

```
PS > 'Bahnhof' -match '\bhof'
False
PS > 'Ein Hof' -match '\bhof'
True
```

Allerdings muss das Suchwort bislang nur an einer Wortgrenze beginnen und kann deshalb der Anfang eines längeren Worts sein:

```
PS > 'Ein Hoffnungsschimmer' -match '\bhof'
True
```

Setzen Sie auch an das Ende des Textmusters einen Anker, wenn es als eigenständiges Wort erkannt werden soll:

```
PS > 'Ein Hoffnungsschimmer' -match '\bhof\b'  
False  
PS > 'Ein Hof' -match '\bhof\b'  
True
```

Der *-match*-Operator unterscheidet, wie alle Standardoperatoren, nicht zwischen Groß- und Kleinschreibung. Wollen Sie diese Unterscheidung nutzen, setzen Sie die »c«-Variante des Operators ein, also *-cmatch*:

```
PS > 'Ein Hof' -match '\bhof\b'  
True  
PS > 'Ein Hof' -cmatch '\bhof\b'  
False  
PS > 'Ein hof' -cmatch '\bhof\b'  
True
```

Die Unterscheidung zwischen Groß- und Kleinschreibung kann auch in die Beschreibung des Textmusters integriert werden. Praktisch ist dies, wenn Sie bei Teilen des Textmusters die Groß- und Kleinschreibung beachten wollen, bei anderen aber nicht. Mit dem Ausdruck »(?-i)« schalten Sie die Unterscheidung zwischen Groß- und Kleinschreibung ein und mit »(?i)« schalten Sie die Unterscheidung aus. Das folgende Textmuster findet alle Dateinamen, die mit einem kleinen »a« beginnen und als Dateierweiterung *.txt* in beliebiger Groß- und Kleinschreibung aufweisen.

```
PS > 'Abc.txt' -match '(?-i)a.*(?i)(\.txt)+'  
False  
PS > 'abc.txt' -match '(?-i)a.*(?i)(\.txt)+'  
True  
PS > 'abc.TXT' -match '(?-i)a.*(?i)(\.txt)+'  
True
```

Informationen aus einem Text extrahieren

Sie interessieren sich nur für eine bestimmte Information innerhalb eines längeren Texts.

Lösung

Wenn Sie wissen, an welcher Position im Text die gesuchte Information steht, verwenden Sie *SubString()* und geben die Startposition und Länge an:

```
PS > 'Dieser Text enthält eine wichtige Information'.SubString(7, 4)  
Text
```

Suchen Sie ein bestimmtes Textmuster innerhalb des Texts, verwenden Sie reguläre Ausdrücke zusammen mit dem PowerShell-Operator `-match`. Das folgende Beispiel extrahiert eine E-Mail-Adresse aus einem Text:

```
PS > 'Die E-Mail-Adresse lautet tobias.weltner@scriptinternals.de. Für Angebote →
    verwenden Sie sales@scriptinternals.de' -match '\b[A-Z0-9._%+-]+\b'
True
```

Die extrahierte E-Mail-Adresse steht danach in der Variablen `$matches` zur Verfügung:

```
PS > $matches[0]
tobias.weltner@scriptinternals.de
```

Gibt es im Text ein auffälliges Trennzeichen, können Sie den Text mit `Split()` auch in ein Textfeld (Array) aufsplitten und danach das gewünschte Bruchstück ansprechen. Die folgende Codezeile extrahiert aus einem Pfadnamen den Dateinamensanteil und verwendet als Trennzeichen den in Pfadnamen üblichen umgekehrten Schrägstrich.

Weil das Ergebnis ein Array ist, können Sie mit eckigen Klammern auf ein bestimmtes Element daraus zugreifen. Der Index `-1` entspricht dem letzten Arrayelement, also in diesem Beispiel dem Dateinamen:

```
PS > "c:\windows\system32\test.dll".Split('\')
c:
windows
system32
test.dll
PS > "c:\windows\system32\test.dll".Split('\')[-1]
test.dll
```

Entsprechend finden Sie mit nur einer kleinen Änderung auch die Dateierweiterung heraus. Ersetzen Sie dazu einfach das Trennzeichen durch den Punkt und greifen Sie mit dem Index `-1` erneut auf das letzte Element des Arrays zu:

```
PS > "c:\windows\system32\test.dll".Split('.')[-1]
dll
```

`Split()` ist eine ausgesprochen vielseitige Form der Stringanalyse. Sie können die Methode beispielsweise auch dazu verwenden, um komma- oder tabulatorseparierte Informationen in ihre Einzelteile auseinanderzuberechnen:

```
PS > "Tom,Peter,Michael,Marie,Andrea".Split(',')
Tom
PS > "Tom,Peter,Michael,Marie,Andrea".Split(',')[-1]
Andrea
```

Das Tabulatorzeichen wird in PowerShell entweder als `"`t"` oder als `([char]9)` angegeben.

Hintergrund

Befinden sich die gesuchten Informationen an einer festgelegten Stelle im Text, verwenden Sie *SubString()*, um den betreffenden Teil des Texts auszulesen. Dabei geben Sie lediglich an, ab welcher Textposition Sie wie viele Zeichen lesen möchten. Geben Sie die Länge des zu lesenden Texts nicht an, wird der gesamte Rest des Texts gelesen:

```
PS > 'Hallo Welt'.SubString(3,5)
lo We
PS > 'Hallo Welt'.SubString(3)
lo Welt
```

Häufig genügt diese einfache Form der Textextraktion. Interessieren Sie sich für die Dateierweiterung eines Dateinamens, könnten Sie die letzten drei Zeichen des Pfads abschneiden:

```
PS > dir $env:windir | ForEach-Object { $_.Name.Substring($_.Name.Length-3) }
```

Dieser Ansatz würde allerdings nur bei Dateierweiterungen funktionieren, die genau drei Zeichen lang sind. Bei Ordnern wäre das Ergebnis verwirrend. Präziser wäre es, wenn Sie die Position des letzten Punkts im Pfad bestimmen und von dort ab alle Zeichen ausgeben. Sorgen Sie außerdem dafür, dass keine Ordner bearbeitet werden:

```
PS > dir $env:windir | Where-Object { ($_.PSIsContainer -eq $false) } | →
    ForEach-Object { $_.Name.Substring($_.Name.LastIndexOf('.')) }
.jpg
.jpg
.txt
.exe
(...)
```

Jetzt allerdings erhalten Sie Fehlermeldungen für alle Dateien, die keine Erweiterung tragen, und müssten den Fehler abfangen:

```
PS > dir $env:windir | Where-Object { ($_.PSIsContainer -eq $false) } | →
    ForEach-Object { trap {'(keine)'; Continue} →
        $_.Name.Substring( $_.Name.LastIndexOf('.')) }
```

Sofern Sie mit Datei- und Ordnerobjekten arbeiten, steht die Erweiterung allerdings auch direkt zum Abruf bereit:

```
PS > dir $env:windir | ForEach-Object { $_.Extension }
```

Einfacher funktioniert in solchen Fällen die Methode *Split()*, mit der Sie lediglich ein Trennzeichen angeben, das im Text vorkommt. Das Ergebnis ist ein an den Trennzeichen jeweils aufgetrennter Text, der als Feld (Array) zurückgeliefert wird.

```
PS > 'a+b+c+d+e+f'.Split('+')
a
b
c
d
e
f
```

Ab PowerShell Version 2 steht zusätzlich der Operator *-split* zur Verfügung, der allerdings als Trennzeichen einen regulären Ausdruck verlangt und deshalb anfangs womöglich nicht so funktioniert wie erwartet:

```
PS > "c:\windows\system32\test.dll" -split '.'

PS > "c:\windows\system32\test.dll" -split '\'
Fehlerhaftes Argument für den Operator "-split": "\" wird analysiert -
Unzulässiger \ am Ende des Musters..
Bei Zeile:1 Zeichen:38
+ "c:\windows\system32\test.dll" -split <<<< '\'
+ ~~~~~ CategoryInfo          : InvalidOperation: (:) [], RuntimeException
+ ~~~~~ FullyQualifiedErrorId : BadOperatorArgument
```

Richtig wären diese Aufrufe, die das Trennzeichen jeweils zuerst mit dem in regulären Ausdrücken üblichen Escape-Zeichen »\« entwerten:

```
PS > "c:\windows\system32\test.dll" -split '\.'
c:\windows\system32\test
dll
PS > ("c:\windows\system32\test.dll" -split '\.')[1]
dll
PS > "c:\windows\system32\test.dll" -split '\\\'
c:
windows
system32
test.dll
```

TIPP

Der .NET-Typ *Regex* enthält mit der Methode *Escape()* eine einfache Möglichkeit, Text automatisch so umzuwandeln, dass alle reservierten Sonderzeichen entwertet werden:

```
PS > [Regex]::Escape("c:\windows")
c:\windows
```

Alternativ könnten Sie auch den Operator anweisen, keine regulären Ausdrücke zu verwenden. Nun allerdings wird der Aufruf von *-split* deutlich aufwändiger, als direkt die in Strings vorhandene *Split()*-Methode einzusetzen:


```
PS > ("c:\windows\system32\test.dll" -split '.',0,'SimpleMatch')[-1]
dll
PS > "c:\windows\system32\test.dll".split('.')[-1]
dll
```

HINWEIS Der Operator *-split* ist ein äußerst flexibler Mechanismus, um Texte intelligent in Teiltexen aufzusplitten. Beispielsweise darf das Trennzeichen auch ein ausführbarer Skriptblock sein, der dynamisch bestimmt, wann gesplittet werden soll:

```
PS > "Hallo Welt,dies ist:ein Test" -split {[char[]]' ,:.'} -contains $_}
Hallo
Welt
dies
ist
ein
Test
```

In diesem Fall splittet der Operator den Text an allen Trennzeichen, die in der Liste angegeben wurden, also an Leerzeichen, Kommata und Doppelpunkten. Ähnliches kann man auch über reguläre Ausdrücke erreichen und hat dabei sogar die Möglichkeit, das für die Trennung herangezogene Zeichen weiterhin im Ergebnistext zu belassen. Das folgende Beispiel trennt einen Text in Sätze auf. Getrennt wird nach den im regulären Ausdruck aufgeführten Satzendezeichen. Diese Zeichen werden nicht aus dem Text entfernt.

```
PS > $string = 'Wo ist die Katze? Die Katze liegt auf der Matratze. →
    Nimm die Katze von der Matratze!'
PS > $string -split '\s*(?<=[\.\?!]).+?\s*'
Wo ist die Katze?
Die Katze liegt auf der Matratze.
Nimm die Katze von der Matratze!
```

Mehr zu diesem vielseitigen Operator erfahren über Sie diese Zeile:

```
PS > Get-Help about_split
```

Wenn Sie mit regulären Ausdrücken arbeiten, müssen Sie mit den Platzhalterzeichen der regulären Ausdrücke das gesuchte Muster genau beschreiben (Tabelle 1.6 auf Seite 42). Eine Dateierweiterung eines Pfads könnten Sie beispielsweise folgendermaßen finden:

```
PS > $pfad = 'c:\test\unterordner\datei.txt'
PS > $muster_erweiterung = '\.[A-Z0-9]{1,6}\Z'
PS > $pfad -match $muster_erweiterung
True
PS > $matches[0]
.txt
```

Um das Laufwerk zu extrahieren, ändern Sie nun lediglich die Definition des gesuchten Textmusters:

```
PS > $pfad = 'c:\test\unterordner\datei.txt'
PS > $muster_laufwerk = '\A[A-Z0-9]{1,6}:'
PS > $pfad -match $muster_laufwerk
True
PS > $matches[0]
c:
```

TIPP

Weil Pfadnamen sehr häufig in ihre Bestandteile zerlegt werden müssen, gibt es hierfür einen schnelleren und eleganteren Weg, nämlich das Cmdlet *Split-Path*:

```
PS > $pfad = 'c:\test\unterordner\datei.txt'
PS > Split-Path $pfad -leaf
datei.txt
PS > Split-Path $pfad -noQualifier
\test\unterordner\datei.txt
PS > Split-Path $pfad -parent
c:\test\unterordner
PS > Split-Path $pfad -qualifier
c:
```

Mehrere Treffer finden

Der *-match*-Operator liefert stets nur das erste gefundene Textmuster zurück. Sie erhalten in diesem Fall also nur die erste E-Mail-Adresse zurück:

```
PS > $text = 'Wollen Sie hervorragende PowerShell-Inhouse-Kurse organisieren? →
Über tobias.weltner@scriptinternals.de stehe ich gern als Trainer zur →
Verfügung. Für Angebote verwenden Sie sales@scriptinternals.de'
PS > $email = '\b[A-Z0-9._%+-]+@[A-Z0-9_-]+\.[A-Z]{2,4}\b'
PS > if ($text -match $email) { $matches[0] }
tobias.weltner@scriptinternals.de
```

Wollen Sie das Textmuster mehrfach finden, greifen Sie auf den Typ *Regex* von .NET Framework zurück. Weil die Methoden dieses Typs normalerweise die Groß- und Kleinschreibung unterscheiden, erweitern Sie das Textmuster und stellen »(?*i*)« voran, um Groß- und Kleinschreibung nicht unterscheiden zu lassen:

```
PS > $text = 'Wollen Sie hervorragende PowerShell-Inhouse-Kurse organisieren? Über →
tobias.weltner@scriptinternals.de stehe ich gern als Trainer zur Verfügung. →
Für Angebote verwenden Sie sales@scriptinternals.de'
PS > $email = '(?i)\b[A-Z0-9._%+-]+@[A-Z0-9_-]+\.[A-Z]{2,4}\b'
PS > $regex = [Regex]$email
PS > $regex.Matches($text) | Select-Object -expandProperty Value
tobias.weltner@scriptinternals.de
sales@scriptinternals.de
```

Textinhalte durch andere Texte ersetzen

Sie wollen einen Namen oder eine andere Information innerhalb eines Texts ersetzen.

Lösung

Verwenden Sie den Operator *–replace*:

```
PS > 'Herr Müller war gestern hier' -replace 'Müller', 'Meier'  
Herr Meier war gestern hier
```

Hintergrund

Der Operator *–replace* ersetzt das gefundene Wort überall in einem Text. Im einfachsten Fall geben Sie dazu lediglich das Such- und das Ersetzungswort an. Allerdings wird hier das Suchwort auch dann ersetzt, wenn es Teil eines anderen Worts ist:

```
PS > 'Herr Müllerjahn war gestern hier' -replace 'Müller', 'Meier'  
Herr Meierjahn war gestern hier
```

Die Ersetzung funktioniert deshalb präziser, wenn Sie das Suchwort als regulären Ausdruck genauer beschreiben. Suchen Sie nach einem Wort, fügen Sie beispielsweise Textanker hinzu (Tabelle 1.8 auf Seite 44) und legen so fest, dass das Suchwort an einer Wortgrenze beginnen und enden muss:

```
PS > 'Herr Müllerjahn war gestern hier. Herr Müller auch.' →  
-replace '\bMüller\b', 'Meier'  
Herr Müllerjahn war gestern hier. Herr Meier auch.
```

Mit Textankern lassen sich auch Markierungen am Textanfang oder Textende hinzufügen, beispielsweise um einen Text als E-Mail-Antwort zu markieren. Der Anker für den Satzanfang lautet »^«:

```
PS > 'Herr Müllerjahn war gestern hier. Herr Müller auch.' -replace '^', '>>> '  
>>> Herr Müllerjahn war gestern hier. Herr Müller auch.
```

Bei mehrzeiligen Texten verweist dieser Anker allerdings zunächst immer auf den Textanfang. Mehrere Sätze werden also von *–match* nicht automatisch als individuelle Sätze gewertet:

```

PS > $text = @'
>> Herr Müller war gestern hier.
>> Er kam unerwartet.
>> Wir haben Herrn Müller Ihre E-Mail-Adresse gegeben.
>> '@
>>
PS > $text -replace '^', '>>>'
>>>Herr Müller war gestern hier.
>>>Er kam unerwartet.
>>>Wir haben Herrn Müller Ihre E-Mail-Adresse gegeben.
Damit die einzelnen Sätze separat ausgewertet und Ihr Zusatz vor jeden Satzanfang gestellt
wird, schalten Sie im regulären Ausdruck mit »(?m)« den Multiline-Modus ein:
PS > $text = @'
>> Herr Müller war gestern hier.
>> Er kam unerwartet.
>> Wir haben Herrn Müller Ihre E-Mail-Adresse gegeben.
>> '@
>>
PS > $text -replace '(?m)^\s', '>>>'
>>>Herr Müller war gestern hier.
>>>Er kam unerwartet.
>>>Wir haben Herrn Müller Ihre E-Mail-Adresse gegeben.

```

Reguläre Ausdrücke erlauben sogenannte Rückverweise. Damit wird es möglich, bei der Ersetzung auf das ursprüngliche Stichwort Bezug zu nehmen. Möchten Sie zum Beispiel eine Liste mit PC-Namen so verändern, dass zwischen »PC« und laufender Nummer ein weiterer Begriff eingefügt wird, könnten Sie dies mithilfe der Rückverweise realisieren.

Generieren Sie zuerst als Ausgangsmaterial eine einfache PC-Liste:

```

PS > 1..100 | ForEach-Object { "PC$ " } | Out-File $env:temp\pcliste.txt
PS > Invoke-Item $env:temp\pcliste.txt

```

Diese Liste soll nun so bearbeitet werden, dass die Zahl hinter »PC« durch eine dreistellige Zahl ersetzt wird. Hier die Lösung:

```

PS > Get-Content $env:temp\pcliste.txt | ForEach-Object {
    { $_ -replace '\bPC(\d*)\b', 'PC-W7-$1' } | Out-File $env:temp\pcliste2.txt
}
PS > Invoke-Item $env:temp\pcliste2.txt

```

In diesem Beispiel werden *–replace* wieder zwei Informationen zur Verfügung gestellt: das Suchmuster sowie der Ersetzungsbegriff. Das Suchmuster definiert ein Muster, das aus dem festen Text »PC« sowie einer beliebig langen Zahl besteht (»\d*«). Gesucht wird das Muster nicht innerhalb eines Worts, sondern als eigenständiges Wort (Wortgrenzen: »\b«).

Der Ersetzungsbegriff besteht aus dem festen Text »PC-W7-« sowie dem Rückverweis »\$1«. Dieser Rückverweis sieht zwar eigentlich aus wie eine PowerShell-Variable, ist aber keine. Deshalb darf dieser Ersetzungstext keinesfalls in doppelte Anführungszeichen gestellt werden, weil PowerShell andernfalls den Rückverweis als Variable interpretieren würde.

Der Rückverweis entspricht dem in Klammern gesetzten Teil des gesuchten Musters, also der Zahl. Die Zahl des Originalbegriffs kann auf diese Weise in den Ersetzungsbegriff übernommen werden.

Tatsächlich kann es beinahe beliebig viele Rückverweise geben. »\$0« steht zum Beispiel für das gesamte gefundene Muster, sodass Sie die Liste auch so bearbeiten könnten:

```
PS > Get-Content $env:temp\pcliste.txt | ForEach-Object →
    { $_ -replace '\bPC(\d*)\b', 'PC-W7-$1 (war $0)' } →
    | Out-File $env:temp\pcliste2.txt
PS > Invoke-Item $env:temp\pcliste2.txt
```

Das Ergebnis lässt sich dabei sogar in anderen Formaten speichern, beispielsweise als automatisch generierte Ersetzungstabelle:

```
PS > $1 = @('Name Neu,Name alt')
PS > $1 += Get-Content $env:temp\pcliste.txt | ForEach-Object →
    { $_ -replace '\bPC(\d*)\b', '"PC-W7-$1","$0"' }
PS > $1 | Out-File $env:temp\pcliste2.csv
PS > Import-CSV $env:temp\pcliste2.csv | Export-CSV →
    $env:temp\pcliste3.csv -useCulture -NoTypeInfoation →
    -Encoding UTF8
PS > Invoke-Item $env:temp\pcliste3.csv
```

Leerzeichen aus Texten entfernen

Sie möchten überflüssige Leerzeichen aus Texten entfernen.

Lösung

Führende oder abschließende Leerzeichen entfernen Sie mit der String-Methode *Trim()*:

```
PS > '   Dieser Text enthält überflüssige Leerzeichen   '.Trim()
Dieser Text enthält überflüssige Leerzeichen
```

Möchten Sie nur führende Leerzeichen entfernen, verwenden Sie *TrimStart()*. *TrimEnd()* entfernt Leerzeichen am Ende des Texts.

```
PS > '   Dieser Text enthält überflüssige Leerzeichen   '.TrimStart()
Dieser Text enthält überflüssige Leerzeichen
PS > '   Dieser Text enthält überflüssige Leerzeichen   '.TrimEnd()
Dieser Text enthält überflüssige Leerzeichen
```

Wollen Sie überflüssige Leerzeichen innerhalb des Texts entfernen, setzen Sie den *-replace*-Operator zusammen mit regulären Ausdrücken ein. Das folgende Beispiel ersetzt zwei oder mehr aufeinander folgende Leerzeichen durch eines:

```
PS > 'Dieser Text enthält zu viele Leerzeichen.' -replace '\s{2,}', ' '
Dieser Text enthält zu viele Leerzeichen.
```

Hintergrund

Häufig genügt es, Leerzeichen am Anfang oder am Ende eines Texts zu entfernen. Hierzu dienen *Trim()*, *TrimStart()* und *TrimEnd()*.

HINWEIS

Alle *Trim*-Methoden akzeptieren optional auch Argumente, die aus den Zeichen bestehen, die als überflüssig angesehen werden. Die Methoden können also nicht nur Leerzeichen entfernen, sondern beliebige Zeichen. Die folgende Zeile würde am Textanfang und -ende jeweils alle Zeichen entfernen, die als Argument an die *Trim*-Methode übergeben wurde. Innerhalb des Texts werden die Zeichen nicht entfernt, wodurch sich die *Trim*-Methoden von den *Replace*-Methoden unterscheiden:

```
PS > " ../xx Hier steht der Text ....".Trim('./x ')
Hier steht der Text
```

Aber auch innerhalb des Texts lassen sich Leerzeichen entfernen. Dazu müssen die Leerzeichen mit einem Textmuster über reguläre Ausdrücke identifiziert werden. Der Platzhalter »\s« steht für genau ein Leerzeichen. Wollen Sie zwei oder mehr aufeinander folgende Leerzeichen finden, lautet der reguläre Ausdruck dafür »\s{2,}«.

Mit *-replace* finden Sie das gesuchte Textmuster nicht nur, sondern können an seiner Stelle einen Ersatztext angeben. In diesem Fall werden also zwei oder mehr aufeinander folgende Leerzeichen durch genau ein Leerzeichen ersetzt.

Auf ähnliche Weise können Sie nun auch andere Merkmale des Texts normalisieren. Möchten Sie zum Beispiel, dass vor einem Punkt kein Leerzeichen steht, wäre dies eine Lösung:

```
PS > 'Vor dem Punkt soll kein Leerzeichen stehen .' -replace '\s+\.', '.'
Vor dem Punkt soll kein Leerzeichen stehen.
```

Möchten Sie dafür sorgen, dass nach jedem Komma genau ein Leerzeichen folgt, gehen Sie so vor:

```
PS > 'Nach jedem Komma, das ich einfüge, soll genau ein Leerzeichen stehen' -replace '\s{2,}', ' '
Nach jedem Komma, das ich einfüge, soll genau ein Leerzeichen stehen
```

Schwieriger ist es, dafür zu sorgen, dass hinter jedem Komma tatsächlich ein Leerzeichen folgt. Das gesuchte Textmuster ist ein Komma, dem ein »Nicht-Leerzeichen« folgt: »,S«. Wenn Sie dieses Textmuster allerdings durch ein Komma mit folgendem Leerzeichen ersetzen, schneiden Sie den auf das Komma folgenden Buchstaben ab, weil er Teil des gesuchten Textmusters war:

```
PS > 'Nach jedem Komma, das ich einfüge, soll ein Leerzeichen stehen.' -replace '\s+', ' '
Nach jedem Komma, as ich einfüge, oll ein Leerzeichen stehen.
```

Wenn Sie Teile des gesuchten Textmusters erhalten wollen, stellen Sie diesen Teil in runde Klammern und machen Sie so einen Unterausdruck daraus. Der Inhalt der Unterausdrücke steht Ihnen danach in den Variablen $\$1$ bis $\$n$ zur Verfügung. Weil diese Variablen keine PowerShell-Variablen sind, dürfen Sie den Ersetzungstext deshalb nicht in doppelte Anführungszeichen stellen. Andernfalls würde PowerShell die Variablen auflösen, was hier natürlich unerwünscht ist.

```
PS > 'Nach jedem Komma,das ich einfüge,soll ein Leerzeichen stehen.' →  
-replace ',(\S)', ', $1'  
Nach jedem Komma, das ich einfüge, soll ein Leerzeichen stehen.
```

Doppelte Wörter entfernen

Sie möchten doppelt vorkommende Wörter aus einem Text entfernen.

Lösung

Verwenden Sie den *-replace*-Operator und identifizieren Sie die doppelt vorkommenden Wörter über einen regulären Ausdruck:

```
PS > 'In diesem diesem Text kommen kommen Wörter doppelt vor' →  
-replace '\b(\w+)(\s+\1){1,}\b', '$1'  
In diesem Text kommen Wörter doppelt vor
```

Hintergrund

Um doppelt vorkommende Wörter zu identifizieren, verwendet der reguläre Ausdruck sogenannte Rückverweise, denn Sie wollen ja nicht ein bestimmtes doppelt vorkommendes Wort finden, sondern jedes doppelt vorkommende Wort.

Der reguläre Ausdruck sucht innerhalb einer Wortgrenze (`»\b«`) zunächst nach einem beliebigen Wort (`»\w+«`). Dieser Ausdruck steht in runden Klammern und wird zu einem Unterausdruck, auf den Sie nun Bezug nehmen können.

Der zweite Teil des regulären Ausdrucks nimmt jetzt auf das erste Wort Bezug, das über `»\1«` angesprochen wird. Gesucht werden also identische folgende Wörter, die durch ein oder mehrere Leerzeichen getrennt sind (`»\s+«`). Gesucht wird mindestens ein identisches folgendes Wort, es können aber auch mehrere sein (`»{1,}«`).

Das Textmuster, das also aus beliebig vielen durch Leerzeichen voneinander getrennten identischen Wörtern besteht, wird durch das erste gefundene Wort ersetzt. Das erste gefundene Wort steht in `»$1«` zur Verfügung, weil Sie es im Textmuster in runde Klammern gesetzt und dadurch zu einem Unterausdruck gemacht haben. Denken Sie daran, dass der Ersetzungstext in einfache und nicht in doppelte Anführungszeichen gestellt werden muss, denn `»$1«` ist keine PowerShell-Variable und darf deshalb nicht von PowerShell automatisch aufgelöst werden.

Wortbereiche finden

Sie wollen einen Textbereich finden, der durch ein Start- und ein Endwort gekennzeichnet ist. Die maximale Größe des Textbereichs soll festlegbar sein.

Lösung

Verwenden Sie den `-match`-Operator zusammen mit einem regulären Ausdruck:

```
PS > 'Einen Wortbereich vom Anfang bis zu seinem Ende ermitteln' -match →  
    '\banfang\W+(?:\w+\W+){1,6}?ende\b'  
True  
PS > $matches[0]  
Anfang bis zu seinem Ende
```

Hintergrund

Um den gesamten Wortbereich von seinem Anfangs- bis zu seinem Endwort zu ermitteln, könnte man im einfachsten Fall diesen regulären Ausdruck verwenden:

```
PS > 'Einen Wortbereich vom Anfang bis zu seinem Ende ermitteln' -match →  
    '\banfang.*ende\b'  
True  
PS > $matches[0]  
Anfang bis zu seinem Ende
```

Allerdings würde dieser Ausdruck nicht das erwartete Ergebnis liefern, wenn die Start- und Ende-Wörter mehrfach vorkommen:

```
PS > 'Einen Wortbereich vom Anfang bis zu seinem Ende ermitteln. Ende.' -match →  
    '\banfang.*ende\b'  
True  
PS > $matches[0]  
Anfang bis zu seinem Ende ermitteln. Ende
```

Tatsächlich versuchen reguläre Ausdrücke immer, den längstmöglichen Treffer zu finden. Dieses Verhalten nennt man auch »gierig« oder »greedy«. Wollen Sie lieber den kürzestmöglichen Treffer finden, weisen Sie den regulären Ausdruck an, »faul« oder »lazy« zu sein. Dazu fügen Sie an den Quantifizierer ein Fragezeichen an:

```
PS > 'Einen Wortbereich vom Anfang bis zu seinem Ende ermitteln. Ende.' -match →  
    '\banfang.*?ende\b'  
True  
PS > $matches[0]  
Anfang bis zu seinem Ende
```


Jetzt findet der reguläre Ausdruck tatsächlich den erwarteten Textbereich. Allerdings ist es dem Ausdruck egal, wie viele Wörter zwischen Anfang und Ende stehen:

```
PS > 'Am Anfang war das Feuer. Wo das alles hinführt, wird man erst am →
Ende wissen.' -match '\banfang.*?ende\b'
True
PS > $matches[0]
Anfang war das Feuer. Wo das alles hinführt, wird man erst am Ende
```

Möchten Sie festlegen, wie viele Wörter maximal zwischen Anfang und Ende stehen dürfen, muss der sehr unspezifische Baustein `».*«` (beliebige Zeichen, beliebig oft) durch ein spezielleres Konstrukt ersetzt werden. Legen Sie also stattdessen fest, wie viele Wörter zwischen den Kennzeichnern maximal stehen dürfen:

```
PS > 'Am Anfang war das Feuer. Wo das alles hinführt, wird man erst am →
Ende wissen.' -match '\banfangW+(\w+W+){1,6}?ende\b'
False
PS > $matches[0]
Anfang war das Feuer. Wo das alles hinführt, wird man erst am Ende

PS > 'Zwischen Anfang und dem Ende dürfen nur maximal 6 Wörter stehen.' →
-match '\banfangW+(\w+W+){1,6}?ende\b'
True
PS > $matches[0]
Anfang und dem Ende
PS > $matches
```

Name	Value
----	-----
1	dem
0	Anfang und dem Ende

Die Wörter zwischen den Anfangs- und Ende-Wörtern sind im regulären Ausdruck in runde Klammern gestellt, also ein weiterer Unterausdruck. Weil Sie diesen Unterausdruck aber nicht separat auswerten wollen, lässt sich Speicherplatz sparen, indem Sie darin `»?:«` an den Anfang stellen. Der reguläre Ausdruck speichert dann das Ergebnis dieses Unterausdrucks nicht in `$matches`.

```
PS > 'Zwischen Anfang und dem Ende dürfen nur maximal 6 Wörter stehen.' -match →
'\banfangW+(?:\w+W+){1,6}?ende\b'
True
PS > $matches[0]
Anfang und dem Ende
PS > $matches
```

Name	Value
----	-----
0	Anfang und dem Ende

Text in Groß- oder Kleinbuchstaben umwandeln

Sie möchten einen Text einheitlich in Groß- oder Kleinbuchstaben umwandeln.

Lösung

Verwenden Sie die Methoden *ToUpper()* und *ToLower()*:

```
PS > 'Hallo Welt'.ToUpper()  
HALLO WELT  
PS > 'Hallo Welt'.ToLower()  
hallo welt
```

Hintergrund

In jeden Text sind Textbefehle integriert, die Sie über einen Punkt ansprechen können. In diesem Kapitel haben Sie bereits einige dieser Befehle kennengelernt, und die Befehle zur Umwandlung in Groß- und Kleinbuchstaben gehören ebenfalls dazu.

Die Umwandlung kann sinnvoll sein, wenn Sie eine einheitliche Darstellung wünschen. Die nächste Zeile listet beispielsweise alle ausführbaren Programme aus dem Systemordner auf und gibt die Dateinamen einheitlich in Kleinbuchstaben aus:

```
PS > Dir $env:windir\system32 -filter *.exe -name | ForEach-Object { $_.ToLower() }  
adaptertroubleshooter.exe  
aitagent.exe  
alg.exe  
appidcertstorecheck.exe  
appidpolicyconverter.exe  
...
```

Texte in einzelne Zeichen umwandeln

Sie möchten einen Text in Buchstaben umwandeln.

Lösung

Konvertieren Sie den Text in ein sogenanntes Character-Array, indem Sie dem Text den Typ *[Char[]]* zuweisen:

```
PS > [Char[]] 'Hallo Welt'  
H  
a  
l  
l  
o  
...  
W  
e  
l  
t  
...
```

Möchten Sie lediglich auf einzelne Zeichen eines Texts zugreifen, interpretieren Sie den Text als Feld (Array). Die folgende Zeile liefert das fünfte Textzeichen (mit der Indexnummer 4, weil der Index bei 0 beginnt):

```
PS > 'Windowsordner'[4]  
o
```

Entsprechend finden Sie das Laufwerk eines Pfadnamens folgendermaßen heraus:

```
PS > 'c:\ordner\datei.txt'[0]  
c
```

Hintergrund

Ein Text besteht technisch gesehen aus einzelnen Buchstaben. Möchten Sie auf diese einzelnen Buchstaben zugreifen, wandeln Sie den Text entweder in ein Feld (Array) des Typs *Char* um oder Sie greifen direkt auf die gewünschten Buchstaben zu.

Weil bei Pfaden normalerweise die ersten drei Buchstaben das Laufwerk bezeichnen, könnten Sie diese ersten drei Zeichen auslesen und mit *-join* zu einem Text neu zusammenfassen, um den Laufwerksanteil zu bestimmen:

```
PS > -join 'c:\ordner\datei.txt'[0..2]  
c:\
```

Auch diese Aufgabe kann in PowerShell auf mehr als eine Art gelöst werden:

```
PS > 'c:\ordner\datei.txt'.Split('\')[0]  
PS > Split-Path 'c:\ordner\datei.txt' -Qualifier
```

Auf diese Weise lassen sich auch zufällig generierte Kennwörter erstellen. Der nächste Code entnimmt aus einer Liste erlaubter Zeichen zufällig 20 Zeichen und erzeugt daraus ein Zufallskennwort:

```
PS > $liste = [Char[]]'abcdefgABCDEFG0123456&%$'  
PS > -join (1..20 | ForEach-Object { Get-Random $liste -count 1 })  
CbA3egDcgc55a064D50F  
PS > -join (1..20 | ForEach-Object { Get-Random $liste -count 1 })  
406Bf2Fa6GE1a1Fe6Bfa  
PS > -join (1..20 | ForEach-Object { Get-Random $liste -count 1 })  
DF%D$EDe1fB4&0B4EDg1
```

TIPP

Durch die Umwandlung eines Texts in einzelne Zeichen lassen sich Texte auch als Binärwerte in die Registrierungsdatenbank schreiben. Die Zeichen lassen sich nämlich einfach weiterkonvertieren in ein Byte-Array.

Der folgende Code legt einen Schlüssel in *HKEY_CURRENT_USER\Software\Test* an und legt darin zwei Werte namens »wert1« und »wert2« an. Der erste Wert enthält den Text als normalen Textwert. Der zweite speichert ihn binär als Byte-Array:

```
New-Item Registry::HKEY_CURRENT_USER\Software\Test -ErrorAction SilentlyContinue | Out-Null
Set-ItemProperty Registry::HKEY_CURRENT_USER\Software\Test -Name Wert1 -Value 'Hallo Welt'
Set-ItemProperty Registry::HKEY_CURRENT_USER\Software\Test -Name Wert2 -Value
([Byte[]][Char[]]'Hallo Welt') -Type Binary
regedit.exe
```

Texte aus mehreren Einzeltexten zusammensetzen

Sie möchten mehrere einzelne Texte zu einem Gesamttext zusammenfügen.

Lösung

Sofern es sich bei allen Bestandteilen aus Texten handelt, könnten Sie den mathematischen Operator »+« verwenden:

```
PS > 'Erster Text' + 'Zweiter Text'
Erster TextZweiter Text
```

Dieser Ansatz schlägt allerdings fehl, wenn der erste Bestandteil kein Text ist, weil PowerShell den Datentyp der gesamten Operation nach dem ersten Bestandteil auswählt. Zuverlässiger und sehr viel vielseitiger funktioniert der Operator `-join`:

```
PS > -join ('Erster Text','Zweiter Text')
Erster TextZweiter Text
```

Sie können jetzt auch das Trennzeichen angeben, das `-join` verwendet, um die Texte miteinander zu verbinden. Normalerweise wird kein Trennzeichen verwendet, sodass die Texte direkt hintereinander angefügt werden. Die folgenden Zeilen demonstrieren abweichende Trennzeichen:

```
PS > ('Erster Text','Zweiter Text') -join ', '
Erster Text, Zweiter Text
PS > ('Erster Text','Zweiter Text') -join "`n"
Erster Text
Zweiter Text
```

Hintergrund

Der mathematische Operator »+« kann zwar Textteile zu einem Gesamttext zusammenfügen, allerdings nur, wenn der erste (links stehende) Ausdruck ein Text ist. Deshalb schlägt dieser Ansatz fehl:

```
PS > $ergebnis = 12
PS > $ergebnis + ' lautet das Ergebnis'
Der Wert " lautet das Ergebnis" kann nicht in den Typ "System.Int32" konvertiert werden.
Fehler: "Die Eingabezeichenfolge hat das falsche Format."
```

Wesentlich einfacher funktioniert der Operator *-join*, der ein Feld (Array) erwartet und dieses dann zuerst falls nötig in Texte verwandelt und danach ausgibt:

```
PS > -join ($ergebnis, ' lautet das Ergebnis')
12 lautet das Ergebnis
```

Die Informationen, die Sie dem Operator übergeben, dürfen auch von anderen Befehlen stammen. Die folgende Zeile liefert die Namen aller laufenden Prozesse:

```
PS > -join (Get-Process | Select-Object -expandProperty Name)
AcroRd32algAppleMobileDeviceServiceApplicationUpdaterconhostcsrsscrrssDataCardMonitordllh
ostdwmesClientexplorerfdmFlashUtil10k_ActiveXIdleiexploreiexploreiPodServiceiTunesHelperL
ightsOutClientServiceIsasslsmdn...
```

Das Ergebnis ist allerdings kaum lesbar, weil die Namen der Prozesse nur aneinandergehängt werden. Nützlich wird die Ausgabe erst, wenn Sie *-join* außerdem ein Trennzeichen übergeben, beispielsweise ein Komma oder einen Zeilenumbruch:

```
PS > (Get-Process | Select-Object -expandProperty Name) -join ', '
AcroRd32, alg, AppleMobileDeviceService, ApplicationUpdater, conhost, csrss, csrss,
DataCardMonitor, dllhost, dwm, esClient, explorer, fdm...
```

TIPP

Auch Textlisten lassen sich auf diese Weise umformatieren. Wenn Sie beispielsweise über eine Textdatei verfügen, in der PC-Namen untereinander aufgeführt werden, kann daraus leicht eine Datei erstellt werden, die die PC-Namen semikolongetrennt aufführt. Um das folgende Beispiel nachzuvollziehen, legen Sie sich im Windows-Editor von Hand eine kurze Textdatei an, in die Sie pro Zeile einen PC-Namen eintragen, und speichern Sie die Datei. Anschließend wird die Datei konvertiert. Passen Sie in der folgenden Zeile den Pfadnamen der Datei *rechnerliste.txt* an, indem Sie stattdessen den Pfadnamen zu Ihrer eigenen eben angelegten Textdatei ersetzen:

```
(Get-Content c:\rechnerliste.txt) -join ';' | Out-File $env:temp\neueliste.txt; Invoke-
Item $env:temp\neueliste.txt
```

Die neu erstellte Datei wird automatisch geöffnet, sodass Sie das Ergebnis begutachten können.

Möchten Sie Pfadnamen aus Einzelinformationen zusammensetzen, gibt es hierfür ein spezielles Cmdlet namens *Join-Path*. Es sorgt automatisch dafür, dass das Ergebnis ein gültiger Pfad ist, indem die Trennzeichen »\« normalisiert werden:

```
PS > $folder = 'c:\myfolder'
PS > $file = 'file.txt'
PS > Join-Path $folder $file
c:\myfolder\file.txt
```

Möchten Sie größere Texte zu einem Gesamttext zusammenfassen, verwenden Sie aus Performance-Gründen ein *StringBuilder*-Objekt. Der traditionelle Ansatz, Textinformationen Schritt für Schritt mit »+=« einem Gesamttext anzufügen, ist langsam und speicherintensiv:

```
PS > Measure-Command { Dir $env:windir\system32 | ForEach-Object {
    { $gesamt += $_.FullName } }
Days           : 0
Hours          : 0
(..)
TotalSeconds   : 2,8733601
TotalMilliseconds : 2873,3601
```

Wesentlich schneller arbeitet der *StringBuilder* von .NET Framework:

```
PS > $sb = New-Object System.Text.StringBuilder
PS > Measure-Command { Dir $env:windir\system32 | ForEach-Object {
    { [Void]$sb.Append($_.FullName) } }
Days           : 0
Hours          : 0
(..)
TotalSeconds   : 0,7555794
TotalMilliseconds : 755,5794
```

Die beinahe vierfache Geschwindigkeit resultiert aus der Art, wie ein *StringBuilder* Texte erstellt. Anders als beim Operator »+=« wird hierbei nicht ständig ein neuer Gesamttext gebildet. Stattdessen erstellt der *StringBuilder* eine interne Tabelle der Einzeltexte und fügt diese erst dann zu einem Gesamttext zusammen, wenn seine *toString()*-Methode aufgerufen wird:

```
PS > $sb.toString()
```

Texte in eine Datei schreiben

Sie möchten einen Text in eine Datei schreiben.

Lösung

Verwenden Sie das Cmdlet *Out-File*. Möchten Sie den Text an eine Textdatei anhängen, geben Sie zusätzlich den Parameter *-Append* an.

```
PS > 'Hello World' | Out-File $home\testfile.txt
PS > 'Next Line' | Out-File $home\testfile.txt -Append
PS > Invoke-Item $home\testfile.txt
```

Hintergrund

Out-File schreibt Informationen des vorangegangenen Befehls in eine Textdatei. Mit dem Parameter *-Append* kann an eine existierende Datei angehängt werden. Andernfalls werden existierende Dateien ohne weitere Warnung überschrieben. Wünschen Sie eine Warnung, geben Sie den Parameter *-NoClobber* an.

Um das Ausgabeformat der Datei zu bestimmen, geben Sie den Parameter *-Encoding* an und legen das Format fest, beispielsweise *UTF8* oder *UNICODE*.

Leiten Sie die Ergebnisse anderer Cmdlets an *Out-File* weiter, gelten dieselben Breitenbegrenzungen wie in der Konsole. Zu breite Informationen werden deshalb in der Textdatei genauso abgeschnitten, als wären sie in die Konsole ausgegeben worden. Dieses Problem können Sie aber umgehen, indem Sie die folgende Kombination aus *Format-Table* und *Out-File* einsetzen und mit dem Parameter *-Width* die maximale Breite der Textdatei angeben:

```
PS > Get-Process | Format-Table -AutoSize | Out-File $env:temp\list.txt -Width 10000
PS > Invoke-Item $env:temp\list.txt
```

Die maximale Breite kann großzügig bemessen sein, da sie in der Regel nicht ausgeschöpft wird. Die Option *-AutoSize* des Cmdlets *Format-Table* sorgt nämlich dafür, dass die Spalten so eng wie möglich gestellt werden, ohne dabei Informationen abzuschneiden. Informationen werden nun in der Textdatei nur noch dann abgeschnitten, wenn es sich bei den Informationen um mehrzeiligen Text handelt. In diesem Fall erscheint nur die erste Zeile des Texts.

HINWEIS

Wenn Sie die Ergebnisdatei im Editor betrachten, achten Sie darauf, im Menü *Format* den Zeilenumbruch auszuschalten. Andernfalls ist die Darstellung der Datei verzerrt.

Leider existiert kein Cmdlet, um Text ohne Zeilenumbruch an eine Datei anzuhängen. Um das zu erreichen, greifen Sie direkt auf .NET Framework zu. Der folgende Code legt zuerst eine Textdatei an und fügt dann zwei Textzeilen hinzu. Anschließend werden mit *AppendAllText()* einzelne Wörter an die Datei angehängt, ohne dass dabei ein Zeilenumbruch eingefügt wird:

```
PS > "Zeile1" | Out-File $env:temp\datei.txt
PS > "Zeile2" | Out-File $env:temp\datei.txt -Append
PS > "Zeile3" | Out-File $env:temp\datei.txt -Append
PS > [System.IO.File]::AppendAllText("$env:temp\datei.txt", 'Wort 1', →
[System.Text.Encoding]::Unicode)
PS > [System.IO.File]::AppendAllText("$env:temp\datei.txt", 'Wort 2', →
[System.Text.Encoding]::Unicode)
PS > [System.IO.File]::AppendAllText("$env:temp\datei.txt", 'Wort 3', →
[System.Text.Encoding]::Unicode)
PS > Get-Content $env:temp\datei.txt
Zeile1
Zeile2
Zeile3
Wort 1Wort 2Wort 3
```

Achten Sie dabei unbedingt auf das korrekte Encoding. *Out-File* generiert Dateien als Vorgabe im Unicode-Format, während die Low-Level-.NET-Methoden als Standard das Encoding UTF8 verwenden. Wenn Sie also beim Aufruf der .NET-Methoden kein Encoding angeben, wird UTF8-Text in eine Unicode-Textdatei geschrieben. Weil Unicode pro Zeichen zwei Bytes verwendet, UTF8 jedoch nur eins, würde der hinzugefügte UTF8-Text als Unicode interpretiert und sonderbar anmutende chinesische Schriftzeichen produzieren.

HINWEIS

Zwar kann Text auch über die klassischen Umleitungszeichen »>« und »>>« in eine Datei geschrieben bzw. an eine Datei angefügt werden, doch sollten Sie diese Möglichkeit zugunsten von *Out-File* nicht verwenden. Nur *Out-File* bietet Ihnen die Möglichkeit, die wesentlichen Details zu bestimmen, also beispielsweise das Encoding festzulegen. Vorsicht ist auch bei anderen Cmdlets geboten, die Text in Dateien schreiben können. So würde *Set-Content* Objekte nicht auf die PowerShell-typische Weise in Text verwandeln:

```
PS > Get-Process | Set-Content $env:temp\datei1.txt
PS > Get-Process | Out-File $env:temp\datei2.txt
PS > Invoke-Item $env:temp\datei1.txt, $env:temp\datei2.txt
```

Einträge eines Logbuchs zeilenweise auswerten

Sie wollen die Texteinträge in einer Logbuchdatei zeilenweise auswerten und aus den Logbuch-Rohdaten sinnvolle Informationen extrahieren.

Lösung

Lesen Sie die Logbuchdatei entweder mit *Get-Content* zeilenweise ein und bearbeiten Sie die einzelnen Zeilen in einer *ForEach-Object*-Schleife. Die folgende Zeile liest die *windowsupdate.log*-Datei zeilenweise ein und analysiert jede einzelne Textzeile mithilfe von *-match* und einem regulären Ausdruck, um die durch Tabulatorzeichen separierten Informationen aufzuschlüsseln. Ausgegeben wird jeweils das sechste Feld pro Zeile:

```
PS > Get-Content $env:windir\windowsupdate.log | ForEach-Object { if ($_ -match →
'(.*)\t(.*)\t(.*)\t(.*)\t(.*)') { $matches[5] }}
DnldMgr ***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-
4A52D1EFE18D}] *****
DnldMgr * Regulation call complete. 0x00000000
DnldMgr ***** DnldMgr: New download job [UpdateId = {1146F9E6-B059-47AB-A6B5-
D62A5CA5F3A4} ...
DnldMgr Regulation: {7971F918-A847-4430-9279-4A52D1EFE18D} - Update 1146F9E6-B059-47AB-
A6B5- ...
DnldMgr * Update is not allowed to download due to regulation.
```

Alternativ verwenden Sie *Switch*. Diese Anweisung arbeitet schneller, kann aber nicht auf bereits geöffnete Dateien zugreifen. Systemdateien müssen deshalb zuerst kopiert werden:


```
PS > Copy-Item $env:windir\windowsupdate.log $home\windowsupdate.log
PS > Switch -regex -file $home\windowsupdate.log →
    '{(.*?)\t(.*?)\t(.*?)\t(.*?)\t(.*?)' { $matches[5] } }
DnldMgr ***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-
4A52D1EFE18D}] *****
DnldMgr * Regulation call complete. 0x00000000
DnldMgr ***** DnldMgr: New download job [UpdateId = {1146F9E6-B059-47AB-A6B5-
D62A5CA5F3A4} ...
DnldMgr Regulation: {7971F918-A847-4430-9279-4A52D1EFE18D} - Update 1146F9E6-B059-47AB-
A6B5- ...
DnldMgr * Update is not allowed to download due to regulation.
```

Hintergrund

Die Datei *windowsupdate.log*, die in diesem Beispiel ausgewertet wird, befindet sich im Windows-Ordner und enthält Informationen über sämtliche Updates, die Windows empfangen hat. Eine Zeile aus dieser Datei sieht beispielsweise so aus:

```
2011-06-01 15:59:36:844 1212 19e0 Agent * Added update {DA613990-5D81-4A4A-8760-
D75319DB4605} ...
```

Die Einzelinformationen sind in dieser Datei mit Tabulatorzeichen voneinander abgegrenzt. Um die einzelnen Informationen aus dieser Zeile mithilfe von *-match* und regulären Ausdrücken zu extrahieren, verwenden Sie Unterausdrücke, kapseln also jede einzelne Information in runde Klammern. Das Muster hierfür könnte so aussehen:

```
PS > $muster = '(.*?)\t(.*?)\t(.*?)\t(.*?)\t(.*?)\t(.*?)'
```

Weil die einzelnen Informationen innerhalb der Zeile durch Tabulatorzeichen begrenzt sind, kann der reguläre Ausdruck den Platzhalter »\t« verwenden, um die einzelnen Felder voneinander zu trennen. Für jedes Feld verwendet das Muster den simplen Ausdruck »(.*?)«, der beliebig vielen beliebigen Zeichen entspricht. Das Fragezeichen steht für einen »faulen« Ausdruck, der den kürzestmöglichen Treffer liefern soll. Ohne das Fragezeichen könnte dieser Ausdruck ansonsten mehr als ein Informationsfeld enthalten, wenn die Rohzeile mehr Tabulatoren enthält als Ihr Suchmuster.

Natürlich könnten Sie die einzelnen Felder auch präziser beschreiben:

```
PS > $muster = →
    '(\d{4}-\d{2}-\d{2})\t(\d{2}:\d{2}:\d{2}:\d{3})\t(.*?)\t(.*?)\t(.*?)\t(.*?)'
```

Die gefundenen Informationen werden in *\$matches* als Feld zurückgeliefert. Sie greifen auf die Einzelinformationen also über den Index in das Feld zu. Feldelement 0 enthält dabei stets den gesamten Rohtext. Möchten Sie nur die erste Information lesen, also das Datum, greifen Sie auf das zweite Feldelement mit dem Index 1 zu:

```
$matches[1]
```

Weil der Zugriff über Indexzahlen unübersichtlich ist, können Sie den Unterausdrücken Namen zuweisen und den Wert des Unterausdrucks dann über diesen Namen erfragen:

```
PS > $muster = →
'(?<Datum>\d{4}-\d{2}-\d{2})\t(?<Zeit>\d{2}:\d{2}:\d{2}:\d{3}) →
\t(?<ID1>.*)\t(?<ID2>.*)\t(?<Aktion>.*)\t(?<Nachricht>.*)'
```

```
PS > $zeile = "2011-06-01`t15:59:36:844`t1212`t19e0`tAgent`t* Added update →
{DA613990-5D81-4A4A-8760-D75319DB4605}.102 to search result"
```

```
PS > $zeile -match $muster
True
```

```
PS > $matches
```

Name	Value
-----	-----
Aktion	Agent
Datum	2011-06-01
ID1	1212
ID2	19e0
Zeit	15:59:36:844
Nachricht	* Added update {DA613990-5D81-4A4A-8760-D75319DB4605}.102 to search ...
0	2011-06-01 15:59:36:844 1212 19e0 Agent * Added update ...

```
PS > $matches.Nachricht
* Added update {DA613990-5D81-4A4A-8760-D75319DB4605}.102 to search result
```

Möchten Sie eine Logbuchdatei zeilenweise auswerten, lesen Sie im einfachsten Fall den Inhalt der Logbuchdatei zeilenweise mit *Get-Content* und bearbeiten danach jede Zeile einzeln in einer *ForEach-Object*-Schleife. Der Kurzbezeichner für solch eine Schleife lautet »%«:

```
PS > $muster = →
'(?<Datum>\d{4}-\d{2}-\d{2})\t(?<Zeit>\d{2}:\d{2}:\d{2}:\d{3})\t(?<ID1>.* →
\t(?<ID2>.*)\t(?<Aktion>.*)\t(?<Nachricht>.*)'
```

```
PS > $pfad = "$env:windir\windowsupdate.log"
```

```
PS > Get-Content $pfad | ForEach-Object { if ($_ -match $muster) →
{$matches.Nachricht} }
```

```
***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-4A52D1EFE18D}]
***** ...
(...)
```

Da Logbuchdateien sehr umfangreich sein können, kann dieser Ansatz sehr zeit- und ressourcenintensiv sein. Schneller funktioniert die Auswertung mit *Switch*. *Switch* benötigt allerdings exklusiven Zugriff auf die Datei. Handelt es sich um eine System-Logbuchdatei, die in Verwendung ist, muss für die Auswertung zuerst eine Kopie erstellt werden:

```
PS > $muster = →
'(?<Datum>\d{4}-\d{2}-\d{2})\t(?<Zeit>\d{2}:\d{2}:\d{2}:\d{3})\t(?<ID1>.* →
\t(?<ID2>.*)\t(?<Aktion>.*)\t(?<Nachricht>.*)'
```

```
PS > Copy-Item $env:windir\windowsupdate.log $home\windowsupdate.log
```

```
PS > Switch -file $home\windowsupdate.log { Default { if ( $_ -match $muster) →
{ $matches.Nachricht } }}
```

```
***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-4A52D1EFE18D}]
***** ...
(...)
```

Switch ist im Gegensatz zur Variante mit *Get-Content* ungefähr doppelt so schnell und liefert bei jedem Schleifendurchlauf die aktuell gelesene Zeile in *\$_* zurück, die dann mit *-match* ausgewertet wird. Alternativ kann *Switch* selbst reguläre Ausdrücke verarbeiten. Dazu fügen Sie die Option *-regex* hinzu und geben innerhalb von *Switch* das oder die Muster an, die Sie überprüfen wollen. Trifft das Muster auf die aktuelle Zeile zu, erhalten Sie in den geschweiften Klammern dahinter in *\$matches* Zugriff auf die gefundenen Textergebnisse:

```
PS > $muster = →
'(<Datum>\d{4}-\d{2}-\d{2})\t(<Zeit>\d{2}:\d{2}:\d{2}:\d{3})\t(<ID1>.*) →
\t(<ID2>.*)\t(<Aktion>.*)\t(<Nachricht>.*)'
PS > Copy-Item $env:windir\windowsupdate.log $home\windowsupdate.log
PS > Switch -regex -file $home\windowsupdate.log { $muster { $matches.Nachricht } }
***** DnldMgr: Regulation Refresh [Svc: {7971F918-A847-4430-9279-4A52D1EFE18D}]
*****
* Regulation call complete. 0x00000000
***** DnldMgr: New download job [UpdateId = {1146F9E6-B059-47AB-A6B5-
D62A5CA5F3A4}.100] *****
Regulation: {7971F918-A847-4430-9279-4A52D1EFE18D} - Update 1146F9E6-B059-47AB-A6B5-
D62A5CA5F3A4 is "Priority" regulated and can NOT download. Sequence 6337 vs AcceptRate 0.
* Update is not allowed to download due to regulation.
```

Im nächsten Beispiel wird die Update-Logbuchdatei nach Einträgen durchsucht, die vom Update-Agent stammen und Updates installieren. Diese Einträge werden den verschiedenen Update-Arten zugeordnet und aufsummiert:

```
PS > Copy-Item $env:windir\windowsupdate.log $home\windowsupdate.log
PS > $ergebnis = @{
    AutoUpdate=0
    AntiVirus=0
    AntiMalware=0
}
PS > switch -regex -file $home\windowsupdate.log →
{ 'START.*Agent: Install.*AutomaticUpdates' →
{ $ergebnis.AutoUpdate += 1} →
'START.*Agent: Install.*Microsoft Security Essentials' →
{ $ergebnis.AntiVirus += 1} →
'START.*Agent: Install.*Microsoft Antimalware' →
{ $ergebnis.AntiMalware += 1}
}
$ergebnis
```

Name	Value
-----	-----
AntiVirus	11
AutoUpdate	2
AntiMalware	6

Mit wenig zusätzlichem Aufwand lassen sich auch sehr viel ausführlichere Informationen auf diese Weise aus einem Logbuch extrahieren. Das nächste Beispiel ermittelt nicht nur die Anzahl

der empfangenen Updates, sondern auch die Titel der Updates. Die Ergebnisse werden dazu wieder in einem Hashtable gesammelt. Diesmal allerdings enthält das Hashtable drei Felder.

Innerhalb von *Switch* werden nun die Startsignaturen der Update-Installationen gesucht und in *\$id* die entsprechende Update-Kategorie gemerkt. *Switch* prüft außerdem auf das Ende einer Update-Installation und setzt dann die Kategorie auf »nichts«.

Solange *\$id* eine gültige Kategorie ist, speichert *Switch* alle gefundenen Update-Titel in der entsprechenden Kategorie. *\$ergebnis.\$id* greift also auf die entsprechende Kategorie innerhalb der Hashtable *\$ergebnis* zu. Weil dieses Hashtable Felder enthält, können die Titel nun als zusätzliche Feldelemente dem Feld hinzugefügt werden. Auf diese Weise ermittelt PowerShell in wenigen Sekunden nicht nur, wie viele Updates installiert wurden, sondern auch, welche genau es waren:

```
PS > Copy-Item $env:windir\windowsupdate.log $home\windowsupdate.log
PS > $ergebnis = @{
    AutoUpdate=@()
    AntiVirus=@()
    AntiMalware=@()
}
PS > $id = ''
PS > Switch -regex -file $home\windowsupdate.log { →
    'START.*Agent: Install.*AutomaticUpdates' →
    { $id = 'AutoUpdate' } →
    'START.*Agent: Install.*Microsoft Security Essentials' →
    { $id = 'Microsoft Security Essentials' } →
    'START.*Agent: Install.*Microsoft Antimalware' →
    { $id = 'Microsoft Antimalware' } →
    'END.*Agent: Install' →
    { $id = '' } →
    'Title = (?<Nachricht>.*)' →
    { if ($id -ne '') { →
    $ergebnis.$id += $matches.Nachricht } →
    } →
    } →
    "Insgesamt $($ergebnis.AutoUpdate.Count) AutoUpdates empfangen." →
    'Liste der Updates:' →
    $ergebnis.AutoUpdate
```

Insgesamt 12 AutoUpdates empfangen.

Liste der Updates:

Update für Windows 7 für x64-basierte Systeme (KB976902)

Update für den Junk-E-Mail-Filter von Outlook 2007 (KB2492475)

Sicherheitsupdate für Windows 7 für x64-basierte Systeme (KB2485376)

Sicherheitsupdate für Windows 7 für x64-basierte Systeme (KB2393802)

(...)

Zusammenfassung

In diesem Kapitel wurden die folgenden Cmdlets eingesetzt:

Name	Beschreibung
<i>Copy-Item</i>	Kopiert eine Datei oder einen Ordner (ohne Inhalt)
<i>ForEach-Object</i>	Alle Ergebnisse der Pipeline nacheinander bearbeiten
<i>Get-Content</i>	Text zeilenweise aus einer Textdatei lesen
<i>Get-Date</i>	Aktuelles Datum erfragen
<i>Get-Service</i>	Liste sämtlicher Dienste erfragen
<i>Get-WmiObject</i>	Informationen über den Computer vom WMI-Dienst erfragen
<i>Join-Path</i>	Fügt mehrere Texte zu einem Pfadnamen zusammen
<i>Measure-Command</i>	Misst die Ausführungszeit eines Befehls oder Skriptteils
<i>New-Timespan</i>	Berechnet eine neue Zeitdifferenz
<i>Out-File</i>	Schreibt Text in eine Datei
<i>Out-String</i>	Objekte in Text umwandeln
<i>Read-Host</i>	Texteingaben aus der Konsole lesen
<i>Split-Path</i>	Teilt einen Dateipfad in seine Bestandteile und erlaubt zum Beispiel, einzelne Informationen, wie das Laufwerk oder den Dateinamen, daraus zu extrahieren

Tabelle 1.9 Cmdlets in diesem Kapitel

Darüber hinaus haben Sie die folgenden Operatoren kennengelernt:

Operator	Beschreibung
<i>-f</i>	Fügt dynamische Informationen in eine Textschablone ein
<i>-join</i>	Fügt mehrere Texte zu einem Gesamttext zusammen
<i>-like</i>	Sucht nach einem Stichwort in einem Text
<i>-match</i>	Sucht nach einem Stichwort in einem Text und verwendet dazu reguläre Ausdrücke
<i>-replace</i>	Ersetzt Text durch einen anderen Text
<i>-split</i>	Trennt Text an einem vorgegebenen Trennzeichen in mehrere Textteile

Tabelle 1.10 Wichtige Operatoren in diesem Kapitel

Kapitel 2

Datum und Zeit

In diesem Kapitel:

Aktuelles Datum ermitteln	72
Kalenderwoche oder Wochentag bestimmen	72
Dateiname mit Zeitstempel erstellen	79
Deutsches Datumsformat verarbeiten	80
Internationale Datumsformate verarbeiten	80
Kulturneutrale Datumsformate verarbeiten	83
Frei definierbare Datumsformate verarbeiten	85
Datum auf Gültigkeit prüfen	86
Zeitdifferenz ermitteln	87
WMI-Zeitangaben umwandeln	90
Systemticks umwandeln	91
Zusammenfassung	92

Zeitinformationen sind allgegenwärtig in der Systemadministration. Sie melden, wann ein bestimmtes Ereignis eingetreten oder wie alt ein Kennwort oder eine Datei ist. Allerdings ist der Umgang mit Datums- und Zeitinformationen normalerweise nicht einfach. Wie bestimmt man aus zwei Vergleichsdaten die Zeitdifferenz, also das Alter? Und wie könnte man das Alter in Stunden oder in Tagen ausdrücken? In welcher Form müssen Datums- und Zeitinformationen angegeben werden, damit PowerShell diese Information richtig interpretiert?

Glücklicherweise enthält PowerShell für die wichtigsten Aufgaben passende Cmdlets, und darüber hinaus haben Sie Zugriff auf die umfangreichen Datums- und Zeitmethoden von .NET Framework.

Aktuelles Datum ermitteln

Sie wollen das aktuelle Datum ermitteln, zum Beispiel als Grundlage für Zeitvergleiche oder als Basis für einen Zeitstempel als Dateinamen.

Lösung

Verwenden Sie *Get-Date*. Dieses Cmdlet liefert die aktuelle Zeit und das aktuelle Datum.

```
PS > Get-Date  
Sonntag, 13. Februar 2011 14:55:15
```

Hintergrund

Get-Date liefert die aktuelle Systemzeit als Datum und Zeitangabe. Das Cmdlet entspricht damit der statischen .NET-Eigenschaft *Now*:

```
PS > [System.DateTime]::Now  
Sonntag, 13. Februar 2011 14:55:15
```

Das Ergebnis ist immer ein *DateTime*-Objekt, das über weitere Eigenschaften und Methoden verfügt. Die folgende Zeile liefert zum Beispiel die aktuelle Zeit in der Einheit »Systemticks«. Weil Systemticks nur wenigen Millisekunden entsprechen, kann man damit auch kürzeste Zeitabstände erfassen und messen:

```
PS > (Get-Date).Ticks  
633483718769092000
```

Kalenderwoche oder Wochentag bestimmen

Sie möchten aus einem Datum beliebige Zeitinformationen lesen, zum Beispiel die Uhrzeit, die Zeitzone, die Kalenderwoche oder den Wochentag.

Lösung

Möchten Sie nur bestimmte Teile einer Zeitinformation nutzen, zum Beispiel nur das Datum, verwenden Sie *Get-Date* mit dem Parameter *-format*:

```
PS > Get-Date -format 'dd.MM.yy'  
14.02.11
```

Interessieren Sie sich nur für das Datum oder nur für die Zeit, setzen Sie den Parameter *-displayHint* ein:

```
PS > Get-Date -displayHint Date  
Montag, 14. Februar 2011
```

```
PS > Get-Date -displayHint Time  
20:14:24
```

Die aktuelle Kalenderwoche erhalten Sie über den Platzhalter *%W* des Parameters *-UFormat*:

```
PS > $KW = Get-Date -UFormat %W  
PS > "Die aktuelle KW ist $KW"
```

HINWEIS

Die auf diese Weise ermittelte Kalenderwoche »0« entspricht der Kalenderwoche »52« des deutschen Kalendariums.

Wollen Sie nicht das aktuelle Systemdatum verwenden, sondern ein anderes Datum, geben Sie dieses Datum hinter *Get-Date* an:

```
PS > Get-Date '18.3.1912' -format 'D'  
Montag, 18. März 1912
```

Hintergrund

Als Vorgabe liefert *Get-Date* das Systemdatum im ausführlichen Format mit ausgeschriebenem Wochentag. Sie können mit *Get-Date* aber auch andere Datumsangaben in ein *DateTime*-Objekt verwandeln. Das Datum muss in dem Datumsformat Ihres Systems angegeben werden, bei deutschen Systemen also in Deutsch. Möchten Sie auch eine Uhrzeit angeben, geben Sie diese mit einem Leerzeichen getrennt hinter dem Datum an:

```
PS > Get-Date '1.1.2000 16:33:12'  
Samstag, 1. Januar 2000 16:33:12
```

Mit dem Parameter *-format* lässt sich das Ausgabeformat verändern und auf bestimmte Datums- oder Zeitangaben beschränken.

```

PS > Get-Date
Sonntag, 13. Februar 2011 15:00:32

PS > Get-Date -format d
13.02.2011
PS > Get-Date -format MMM
Februar
PS > Get-Date -format gg
n. Chr.

```

Zur Verfügung stehen die vordefinierten Zeit- und Datumsformate aus Tabelle 2.1, die immer aus genau einem Buchstaben bestehen. Zwischen Groß- und Kleinschreibung wird unterschieden.

Platzhalter	Beschreibung	Beispiel
<i>d</i>	Abgekürztes Datum	PS > Get-Date -format 'd' 06.06.2011
<i>D</i>	Ausführliches Datum	PS > Get-Date -format 'D' Montag, 6. Juni 2011
<i>f</i>	Ausführliches Datum und abgekürzte Zeit	PS > Get-Date -format 'f' Montag, 6. Juni 2011 19:18
<i>F</i>	Ausführliches Datum und ausführliche Zeit	PS > Get-Date -format 'F' Montag, 6. Juni 2011 19:18:26
<i>g</i>	Abgekürztes Datum und abgekürzte Zeit	PS > Get-Date -format 'g' 06.06.2011 19:19
<i>G</i>	Abgekürztes Datum und ausführliche Zeit	PS > Get-Date -format 'G' 06.06.2011 19:19:45
<i>m, M</i>	Abgekürzte Monats- und Tagangabe	PS > Get-Date -format 'm' 06 Juni
<i>o, O</i>	Formalisierte Datums- und Zeitangabe	PS > Get-Date -format 'o' 2011-06-06T19:20:51.1120000+02:00
<i>r, R</i>	RFC1123-konforme Datums- und Zeitangabe	PS > Get-Date -format 'R' Mo, 06 Jun 2011 19:21:46 GMT
<i>s</i>	Kulturunabhängige universelle Datums- und Zeitangabe	PS > Get-Date -format 's' 2011-06-06T19:22:21
<i>t</i>	Zeitangabe ohne Sekunden	PS > Get-Date -format 't' 19:22
<i>T</i>	Zeitangabe mit Sekunden	PS > Get-Date -format 'T' 19:23:22
<i>u</i>	Kulturunabhängige Standard-Datums- und Zeitangabe	PS > Get-Date -format 'u' 2011-06-06 19:23:54Z

Tabelle 2.1 Vordefinierte Standardformate für Zeit und Datum

Platzhalter	Beschreibung	Beispiel
<i>U</i>	Ausführliches Datum und ausführliche Zeit ohne Zeitzoneberücksichtigung (UTC)	PS > Get-Date -format 'U' Montag, 6. Juni 2011 17:24:36
<i>y, Y</i>	Monat und Jahr	PS > Get-Date -format 'y' Juni 2011

Tabelle 2.1 Vordefinierte Standardformate für Zeit und Datum (*Fortsetzung*)

Sie können auch eigene Formate definieren und dazu die Platzhalterzeichen aus Tabelle 2.2 einsetzen. Verwenden Sie dabei nur ein einzelnes Platzhalterzeichen, interpretiert PowerShell dies im Zweifel als Standardformat. Die folgende Zeile liefert also nicht die Sekunde, sondern das kulturunabhängige Standardformat:

```
PS > Get-Date -format s
2011-06-06T19:29:58
```

Besteht Ihr Format aus mehr als einem Zeichen, ist die Sache eindeutig und PowerShell versteht das »s« als Platzhalter für Sekunden:

```
PS > Get-Date -format ' s'
12
```

Damit Sie in diesem Fall keine unerwünschten Leerzeichen ausgeben, verwenden Sie bei einstelligen Formaten ein vorangestelltes »%«, wenn Sie es als Platzhalter für ein selbstdefiniertes Format kennzeichnen wollen:

```
PS > Get-Date -format '%s'
12
```

Möchten Sie eigenen Text in die Ausgabe integrieren, stellen Sie diesen in einfache oder doppelte Anführungszeichen:

```
PS > Get-Date -format '"Der Monat lautet: %MMMM"'
Der Monat lautet: Juni
```

Platzhalter	Beschreibung	Beispiel
<i>d</i>	Tag	PS > Get-Date -format '%d' 6
<i>dd</i>	Tag, immer zweistellig	PS > Get-Date -format '%dd' 06
<i>ddd</i>	Abgekürzter Name des Tags	PS > Get-Date -format '%ddd' Fr

Tabelle 2.2 Platzhalterzeichen für selbstdefinierte Datumsformate

Platzhalter	Beschreibung	Beispiel
<i>dddd</i>	Voll ausgeschriebener Name des Tags	PS > Get-Date -format 'dddd' Montag
<i>f</i>	1/10 Sekunde	PS > Get-Date -format '%f' 9
<i>ff</i>	1/100 Sekunde	PS > Get-Date -format 'ff' 88
<i>fff</i>	1/1000 Sekunde (Millisekunde)	PS > Get-Date -format 'fff' 884
<i>ffff</i>	1/10.000 Sekunde	PS > Get-Date -format 'ffff' 8842
<i>fffff</i>	1/100.000 Sekunde	PS > Get-Date -format 'fffff' 88420
<i>ffffff</i>	1/1.000.000 Sekunde	PS > Get-Date -format 'ffffff' 884204
<i>fffffff</i>	1/10.000.000 Sekunde	PS > Get-Date -format 'fffffff' 8842041
<i>F-FFFFFF</i>	Wie <i>f</i> , jedoch werden keine 0-Werte angezeigt	PS > Get-Date -format '%F' 9
<i>g, gg</i>	Ära	PS > Get-Date -format '%g' n. Chr.
<i>h</i>	Stunde im 12-Stunden-Modus	PS > Get-Date -format '%h' 6
<i>H</i>	Stunde im 24-Stunden-Modus	PS > Get-Date -format '%H' 18
<i>hh</i>	Stunde zweistellig im 12-Stunden-Modus	PS > Get-Date -format 'hh' 06
<i>HH</i>	Stunde zweistellig im 24-Stunden-Modus	PS > Get-Date -format 'HH' 18
<i>K</i>	Zeitzoneinformation	PS > Get-Date -format '%K' +02:00
<i>m</i>	Minute	PS > Get-Date -format '%m' 1
<i>mm</i>	Minute zweistellig	PS > Get-Date -format 'mm' 01
<i>M</i>	Monat	PS > Get-Date -format '%M' 6
<i>MM</i>	Monat zweistellig	PS > Get-Date -format 'MM' 06

Tabelle 2.2 Platzhalterzeichen für selbstdefinierte Datumsformate (*Fortsetzung*)

Platzhalter	Beschreibung	Beispiel
<i>MMM</i>	Abgekürzter Monatsname	PS > Get-Date -format 'MMM' Jun
<i>MMMM</i>	Ausgeschriebener Monatsname	PS > Get-Date -format 'MMMM' Juni
<i>s</i>	Sekunde	PS > Get-Date -format '%s' 5
<i>ss</i>	Sekunde zweistellig	PS > Get-Date -format 'ss' 05
<i>t, tt</i>	Vormittags/Nachmittags (in deutscher Kultur ohne Bedeutung)	PS > Get-Date -format '%t' 15
<i>y</i>	Jahr ein- oder zweistellig	PS > Get-Date -format '%y' 11
<i>yy</i>	Jahr zweistellig	PS > Get-Date -format 'yy' 11
<i>yyy</i>	Jahr drei- oder vierstellig	PS > Get-Date -format 'yyy' 2011
<i>yyyy</i>	Jahr vierstellig	PS > Get-Date -format 'yyyy' 2011
<i>yyyyy</i>	Jahr fünfstellig	PS > Get-Date -format 'yyyyy' 02011
<i>z</i>	Zeitversatz der Zeitzone	PS > Get-Date -format '%z' +2
<i>zz</i>	Zeitversatz der Zeitzone zweistellig	PS > Get-Date -format 'zz' +02
<i>zzz</i>	Zeitversatz der Zeitzone in Stunden und Minuten	PS > Get-Date -format 'zzz' +02:00
<i>:</i>	Trennzeichen für Zeit	PS > Get-Date -format 'HH:mm' 18:13
<i>/</i>	Trennzeichen für Datum	PS > Get-Date -format 'dd/MM/yy' 06.06.11
<i>" oder '</i>	Wörtlicher Text	PS > Get-Date -format 'ss" Sekunden und "fff" Millisekunden." 57 Sekunden und 348 Millisekunden.

Tabelle 2.2 Platzhalterzeichen für selbstdefinierte Datumsformate (*Fortsetzung*)

Da *Get-Date* ein *DateTime*-Objekt liefert, kann man auch direkt auf dessen Eigenschaften zugreifen (siehe Tabelle 2.3). Den Index des aktuellen Wochentags findet man beispielsweise in der Eigenschaft *DayOfWeek*:

```
PS > (Get-Date).DayOfWeek
Sunday
PS > [Int](Get-Date).DayOfWeek
0
```

Möchte man beispielsweise bestimmte Aufgaben in einem Skript nur an Werktagen durchführen, kann man sich diesen Index zunutze machen und eine Bedingung formulieren:

```
PS > If ( (1..5) -contains [Int](Get-Date).DayOfWeek) {
>> 'Werktag'
>> } else {
>> 'Wochenende'
>> }
>>
```

Die Bedingung überprüft, ob der aktuelle Index in einem Array mit vorgegebenen Zahlen vorhanden ist (*-contains*). Im Beispiel würde ein Werktag erkannt, wenn der aktuelle Index zwischen 1 und 5 liegt, aber natürlich könnten Sie das Array auch anders formulieren. Das nächste Beispiel führt eine Aufgabe nur montags und mittwochs aus:

```
PS > If ( (1,3) -contains [Int](Get-Date).DayOfWeek) {
>> 'Montag oder Mittwoch'
>> } else {
>> 'anderer Tag'
>> }
>>
```

Dabei nutzt das Beispiel aus, dass hinter dem »sprechenden« Namen des aktuellen Wochentags ein numerischer Zahlenwert hinterlegt ist, der nutzbar wird, wenn man den Index in den Datentyp Integer (*[Int]*) umwandelt. Sie könnten aber auch die sprechenden Namen verwenden:

```
PS > If ( ('Saturday', 'Sunday') -contains (Get-Date).DayOfWeek) {
>> 'Wochenende'
>> } else {
>> 'Werktag'
>> }
>>
```

Eigenschaft	Datumsinformation
<i>Date</i>	Datum
<i>Day</i>	Tag
<i>DayOfWeek</i>	Index des Wochentags
<i>DayOfYear</i>	Index des Tags im Jahr

Tabelle 2.3 Eigenschaften eines *DateTime*-Objekts

Eigenschaft	Datumsinformation
<i>Hour</i>	Stunde
<i>Kind</i>	Typ des Datums (zum Beispiel <i>local</i>)
<i>Millisecond</i>	Millisekunde
<i>Minute</i>	Minute
<i>Month</i>	Monat
<i>Second</i>	Sekunde
<i>Ticks</i>	Anzahl Systemticks
<i>TimeOfDay</i>	Zeit
<i>Year</i>	Jahr
<i>DateTime</i>	Datum und Zeit

Tabelle 2.3 Eigenschaften eines *DateTime*-Objekts (*Fortsetzung*)

Dateiname mit Zeitstempel erstellen

Sie möchten einen Dateinamen mit einem aktuellen Zeitstempel erstellen, zum Beispiel als temporären Dateinamen für Logbuchausgaben.

Lösung

Erzeugen Sie den Zeitstempel mit *Get-Date* und seinem Parameter *-Format*, und verwenden Sie den Operator *-f*, um den Zeitstempel in den Pfadnamen einzufügen:

```
PS > $timestamp = Get-Date -Format 'yyyyMMdd-hh-mm-ss'
PS > $path = '{0}\file{1}.txt' -f $env:temp, $timestamp
PS > $path
C:\Users\Tobias\AppData\Local\Temp\file20110213-02-56-27.txt
```

Hintergrund

Get-Date liefert das aktuelle Datum normalerweise in dem in der Systemsteuerung festgelegten Zeitformat. Mit dem Parameter *-Format* können Sie das Format aber auch selbst gestalten. Dazu stehen Ihnen Kürzel zur Verfügung. »yyyy« steht beispielsweise für die vierstellige Jahresangabe. Die Kürzel unterscheiden zwischen Groß- und Kleinschreibung. »MM« steht für den Monat, während »mm« für die Minute steht.

Der auf diese Weise generierte Zeitstempel muss anschließend nur noch in den Pfadnamen eingefügt werden. Diese Aufgabe erledigt der Formatierungsoperator *-f*. Links von ihm steht der Pfadname, den Sie generieren wollen. Darin ist mit »{0}« der Platzhalter vermerkt, an dessen

Stelle der Zeitstempel eingefügt werden soll. Rechts von *-f* wird der Zeitstempel angegeben, der dann an der Position des Platzhalters eingefügt wird.

Deutsches Datumsformat verarbeiten

Sie möchten eine Datumsinformation verwenden, die im deutschen (lokalisierten) Format angegeben wurde.

Lösung

Verwenden Sie *Get-Date* und geben Sie das Datum dahinter als Text an. Der Text muss den gültigen Datumskonventionen Ihres Systems entsprechen:

```
PS > Get-Date 18.3.1912
Montag, 18. März 1912 00:00:00
PS > $geburtstag = Read-Host 'Ihr Geburtstag'
Ihr Geburtstag: 18.3.1912
PS > $datum = Get-Date $geburtstag
PS > 'Ihr Geburtstag fiel auf einen {0:dddd}.' -f $datum
Ihr Geburtstag fiel auf einen Montag.
```

Hintergrund

Datums- und Zeitinformaten werden traditionell in jeder Kultur auf andere Weise formatiert. Damit Sie mit Datums- und Zeitinformaten arbeiten können, muss diese Information also zuerst korrekt interpretiert und dann in einem *DateTime*-Objekt gespeichert werden. Das *DateTime*-Objekt ist kulturunabhängig und speichert die Zeitinformaten immer auf dieselbe vergleichbare Weise.

Liegt das Datum oder die Zeitinformaten in dem Format vor, das in der Systemsteuerung Ihres Computers eingestellt ist, wandeln Sie solche Informationen mit *Get-Date* in ein *DateTime*-Objekt um.

Denken Sie daran, dass Ihr PowerShell-Skript auf anderen Computern mit anderen Ländereinstellungen das Datum im dort gültigen Format erwartet.

Internationale Datumsformate verarbeiten

Sie möchten Datumsinformationen korrekt interpretieren, die nach den Regeln eines anderen Landes formatiert wurden. Sie haben zum Beispiel Daten von einem japanischen Kollegen erhalten und möchten sicherstellen, dass die Datumsinformationen nach japanischen Regeln interpretiert werden.

Lösung

Verwenden Sie die Methode *Parse()* des Typs *DateTime* und geben Sie die Kennung für die Kultur an, aus der das Datum stammt:

```
PS > $datum = '23/2/14 9:47:42'
PS > $kultur = New-Object System.Globalization.CultureInfo("ja-JP")
PS > [DateTime]::Parse($datum, $kultur)
Dienstag, 14. Februar 2023 09:47:42
```

Hintergrund

Die besonderen Eigenarten einer Kultur werden in sogenannten *CultureInfo*-Objekten beschrieben. Möchte man das Datumsformat einer fremden Kultur richtig interpretieren, beschafft man sich das für diese Kultur zuständige *CultureInfo*-Objekt und übergibt es der Methode *Parse()*. Diese verwendet nun die in diesem *CultureInfo*-Objekt enthaltenen Regeln, um das Datum korrekt zu verarbeiten.

CultureInfo-Objekte erhalten Sie unter Angabe des Kulturbezeichners. Für Japan lautet dieser »ja-JP«. Welche Kulturen sonst noch unterstützt werden und wie die Kulturbezeichner lauten, findet diese Zeile für Sie heraus:

```
PS > [System.Globalization.CultureInfo]::GetCultures('SpecificCultures') ->
    | Select-Object DisplayName, Name | Sort-Object DisplayName
```

DisplayName	Name
-----	----
Afrikaans (Südafrika)	af-ZA
Albanien (Albanisch)	sq-AL
Amharisch (Äthiopien)	am-ET
Arabisch (Ägypten)	ar-EG
Arabisch (Algerien)	ar-DZ
Arabisch (Bahrain)	ar-BH
(...)	

Der besondere Vorteil dieses Ansatzes ist, dass das *CultureInfo*-Objekt sämtliche wichtigen Datums- und Zeitformate der Kultur kennt und nicht bloß eines. Wählen Sie beispielsweise die griechische Kultur aus, kann PowerShell alle üblichen Schreibweisen eines griechischen Datums korrekt interpretieren:

```
$daten = @'
14/2/2011
???????, 14 ???????????? 2011
???????, 14 ???????????? 2011 10:08 ?μ
???????, 14 ???????????? 2011 10:08:31 ?μ
14/2/2011 10:08 ?μ
14/2/2011 10:08:31 ?μ
???????, 14 ???????????? 2011 9:08:31 ?μ
'@

PS > $griechisch = New-Object System.Globalization.CultureInfo("el-GR")PS →
> $daten -split "`n" | ForEach-Object { [DateTime]::Parse($_, $griechisch) }

Montag, 14. Februar 2011 00:00:00
Montag, 14. Februar 2011 00:00:00
Montag, 14. Februar 2011 10:08:00
Montag, 14. Februar 2011 10:08:31
Montag, 14. Februar 2011 10:08:00
Montag, 14. Februar 2011 10:08:31
Montag, 14. Februar 2011 09:08:31
```

Umgekehrt lassen sich Datumsinformationen auch im Format anderer Kulturkreise ausgeben. Um das aktuelle Datum in japanischer Schreibweise anzuzeigen, gehen Sie so vor:

```
PS > $japan = New-Object System.Globalization.CultureInfo("ja-JP")
PS > (Get-Date).ToString($japan)
2011/02/14 9:59:49
```

Hier wird das aktuelle Datum gemäß des aktuellen Kalenders in das japanische Format gebracht. In westlichen Kulturen ist der Kalender der Gregorianische Kalender. Andere Kulturen verwenden andere Kalendarien. Um das japanische Datum gemäß des japanischen Kalenders darzustellen, geben Sie den passenden Kalendertyp vor:

```
PS > $japan.DateTimeFormat.Calendar = New-Object →
System.Globalization.JapaneseCalendar
PS > Get-Date
Sonntag, 13. Februar 2011 15:12:32
PS > (Get-Date).ToString($japan)
?? 23/2/13 15:12:06
```

HINWEIS

Enthält die Datumsausgabe Sonderzeichen wie beispielsweise japanische Schriftzeichen, sind diese in der PowerShell-Konsole möglicherweise nicht darstellbar und erscheinen dann als »?«, weil die üblichen Rasterschriftarten der Konsole die Sonderzeichen nicht enthalten. Ändern Sie die Konsolenschriftart in eine TrueType-Schrift, um die Sonderzeichen sichtbar zu machen.

In Unicode-fähigen Skriptumgebungen wie beispielsweise der ISE (»Integrated Script Environment«, Teil der PowerShell 2.0), werden die Zeichen korrekt wiedergegeben.

Die Methode `toString()` akzeptiert die üblichen Formatierungsplatzhalter (siehe Tabelle 2.1 auf Seite 74):

```
PS > $griechisch = New-Object System.Globalization.CultureInfo("el-GR")
PS > 'd','D','f','F','g','G','U' | ForEach-Object { →
    (Get-Date).ToString($_, $griechisch) }
14/2/2011
???????, 14 ???????????? 2011
???????, 14 ???????????? 2011 10:08 ?μ
???????, 14 ???????????? 2011 10:08:31 ?μ
14/2/2011 10:08 ?μ
14/2/2011 10:08:31 ?μ
???????, 14 ???????????? 2011 9:08:31 ?μ
```

Kulturneutrale Datumsformate verarbeiten

Sie möchten Datumsinformationen so verarbeiten, dass die länderspezifischen Einstellungen des Computers keine Rolle spielen und der PowerShell-Code weltweit einsetzbar ist.

Lösung

Setzen Sie das amerikanische Datumsformat ein. Es wird auf allen Systemen in gleicher Weise interpretiert, wenn Sie es direkt in den *DateTime*-Datentyp konvertieren:

```
PS > $datum = [DateTime]'6/18/2011'
PS > $datum
Samstag, 18. Juni 2011 00:00:00

PS > $datum = [DateTime]'6/18/2011 7:30pm'
PS > $datum
Samstag, 18. Juni 2011 19:30:00
```

Oder geben Sie alle Datums- und Zeitangaben in einem formalisierten kulturunabhängigen Format an:

```
PS > Get-Date -Format 'o'
2011-02-14T10:15:58.6442065+01:00
PS > Get-Date -Format 'o' -Date '1.7.2011 18:33:12'
2011-07-01T18:33:12.0000000
```

Liegt das Datum in diesem Format vor, kann es kulturunabhängig auf jedem Rechner wieder eindeutig in ein Datum zurückverwandelt werden:

```
PS > Get-Date '2011-07-01T18:33:12.0000000'
Freitag, 1. Juli 2011 18:33:12
```

Ohne Interpretationsraum und damit stets eindeutig verläuft die Umwandlung auch, indem Sie ein Datum und/oder eine Zeit mit den Parametern von *Get-Date* konstruieren:

```
PS > $weihnachten = Get-Date -day 24 -month 12 -year 2011 -hour 18 -minute 0 →
    -second 0
PS > $weihnachten
Mittwoch, 24. Dezember 2011 18:00:00

PS > Get-Date $weihnachten -format 'o'
2011-12-24T18:00:00.2088656+01:00

PS > 'Es sind noch {0:0} Tage bis Weihnachten.' -f ( $weihnachten - (Get-Date) →
    ).TotalDays
Es sind noch 252 Tage bis Weihnachten.
```

ACHTUNG Geben Sie im letzten Beispiel nicht alle Bestandteile des Datums und der Zeit selbst an, bleiben die nicht von Ihnen festgelegten Bestandteile auf aktuellem Stand. Die folgende Zeile würde also lediglich das Jahr verändern. Alle übrigen Angaben entsprechen der aktuellen Zeit:

```
PS > Get-Date -Year 2012
Dienstag, 14. Februar 2012 10:26:20
```

Hintergrund

Im amerikanischen Kulturkreis werden Daten im Format »Monat/Tag/Jahr« formatiert, und dieses Format verwendet PowerShell auch, wenn Sie einen Text direkt in ein *DateTime*-Objekt verwandeln. Es wird erstaunlicherweise auch »kultureutral« genannt.

Eine direkte Umwandlung eines deutschen Datumstexts in den *DateTime*-Typ stimmen also nicht oder können sogar Fehlermeldungen auslösen, weil PowerShell die erste Angabe immer als Monat wertet:

```
PS > [DateTime]'1.8.2011'
Samstag, 8. Januar 2011 00:00:00

PS > [DateTime]'14.8.2011'
Der Wert "14.8.2011" kann nicht in den Typ "System.DateTime" konvertiert werden. Fehler:
"Die Zeichenfolge wurde nicht als gültiges DateTime erkannt."
Bei Zeile:1 Zeichen:11
+ [DateTime]' <<<< 14.8.2011'
```

Um Skripts international kompatibel zu halten, sollten Sie Datums- und Zeitangaben am besten wie eben gezeigt in einem kulturunabhängigen Format angeben. Bei textbasierten Datums- und Zeitinformationen eignet sich dazu die Datumsformatierung mit dem Format »o«. Legen Sie Datum und Zeit in Ihrem Skript selbst fest, »konstruieren« Sie die Information am besten mit *Get-Date* und seinen Parametern für Zeitangaben.

Parameter	Bedeutung
<code>-date</code>	Parst einen Datumsausdruck. Dies ist der Standardparameter von <i>Get-Date</i> .
<code>-day</code>	Legt den Tag fest
<code>-hour</code>	Legt die Stunde fest
<code>-minute</code>	Legt die Minute fest
<code>-month</code>	Legt den Monat fest
<code>-second</code>	Legt die Sekunde fest
<code>-year</code>	Legt das Jahr fest

Tabelle 2.4 Parameter von *Get-Date* zur Konstruktion von *DateTime*-Objekten

Frei definierbare Datumsformate verarbeiten

Sie möchten eine Datums- oder Zeitinformation in ein *DateTime*-Objekt umwandeln, aber die Information entspricht weder dem lokalen Format noch dem amerikanischen Format. Sie haben zum Beispiel eine Datums- oder Zeitinformation aus einer Logbuchdatei gelesen, die in einem ungewöhnlichen Format vorliegt.

Lösung

Verwenden Sie die *ParseExact()*-Methode des Typs *[DateTime]* und geben selbst an, wie das Format aufgebaut ist, das Sie umwandeln wollen (siehe Tabelle 2.2 auf Seite 75). Die folgende Zeile wandelt einen Text in ein Datum um, der aus dem vierstelligen Jahr, dem Tag und dann dem Monat besteht:

```
PS > $rohform = '20111204'  
PS > $format = 'yyyyMMdd'  
PS > [DateTime]::ParseExact($rohform, $format, $null)  
Sonntag, 4. Dezember 2011 00:00:00
```

Hintergrund

Normalerweise verwendet PowerShell zur Umwandlung von Datums- und Zeitinformationen aus Texten die für die unterschiedlichen Kulturkreise vordefinierten Muster. Weicht die Information von diesen Mustern ab, legen Sie das Muster selbst fest. *ParseExact()* erwartet also das zutreffende Muster, aus dem hervorgeht, an welcher Stelle im Text welche Datums- und Zeitinformationen zu finden sind, und liefert das gewünschte *DateTime*-Objekt zurück.

Datum auf Gültigkeit prüfen

Sie möchten herausfinden, ob ein Text einem gültigen Datumsformat entspricht.

Lösung

Verwenden Sie den Operator `-as` und versuchen Sie, den Text damit in einen *DateTime*-Typ umzuwandeln. Daraus können Sie eine Funktion namens *Check-Date* erstellen, die *\$true* zurückliefert, wenn ein Text einem gültigen deutschen Datum entspricht, andernfalls *\$false*:

```
PS > Function Is-Date($text) { [Bool]($text -as [DateTime]) }
PS > Is-Date Hello
False
PS > Is-Date 1.1.2000
True
```

Wollen Sie ein beliebiges Datumsformat überprüfen, erstellen Sie sich eine Hilfsfunktion wie diese:

```
function Test-Date($text) { $(try { [DateTime]::ParseExact($text, →
'yyyyMMdd', $null)} catch {}) -ne $null }
```

Hier können Sie selbst das Datumsformat hinterlegen, das überprüft werden soll. Die Beispielfunktion prüft, ob es sich bei der Datumsangabe um eine vierstellige Jahreszahl, gefolgt vom Monat und dem Tag handelt.

Hintergrund

Der Operator `-as` entspricht in traditionellen Programmiersprachen dem *TryCast*: Er versucht, Informationen in den angegebenen Typ umzuwandeln. Gelingt dies, ist das Ergebnis der umgewandelte Typ, andernfalls wird nichts zurückgeliefert. Damit eignet sich `-as` für Tests, weil keine Fehlermeldung erscheint, wenn die Umwandlung scheitert.

Die Funktion *Is-Date* macht sich das zunutze. Das Ergebnis von `-as` wird in den Typ *Bool* konvertiert, also einen Ja/Nein-Wert. Liefert `-as` irgendein Ergebnis zurück, konvertiert PowerShell dies zu *\$true*. Liefert `-as` dagegen gar nichts zurück, weil die Umwandlung gescheitert ist, wird der »Nichts«-Wert automatisch in *\$false* umgewandelt.

Sie könnten damit nun Eingabevalidierungen wie diese formulieren:

```
Do { $result = Read-Host 'Datum eingeben' } Until ( Is-Date $result)
```

HINWEIS

Der Operator `-as` berücksichtigt die aktuellen Ländereinstellungen Ihres Systems. Er erwartet also auf einem deutschen System auch das deutsche Datumsformat.

Zeitdifferenz ermitteln

Sie möchten den Zeitunterschied zwischen zwei Zeiten (oder Datumsangaben) ermitteln. Beispielsweise möchten Sie wissen, wie alt in Tagen eine Datei ist, oder Sie möchten messen, wie lange ein bestimmter Vorgang dauert.

Lösung

Verwenden Sie das Cmdlet *New-Timespan*, um ein Datum mit dem aktuellen Datum zu vergleichen. Die folgende Zeile ermittelt das Alter des Windows Explorers in Tagen:

```
PS > (New-Timespan ((Get-Item $env:windir\explorer.exe).CreationTime)).TotalDays
362,627605029067
```

Sie können auch zwei Daten direkt miteinander vergleichen:

```
PS > New-TimeSpan -start '3.12.2010' -end '4.8.2011'

Days           : 244
Hours          : 0
Minutes        : 0
(...)
TotalSeconds   : 21081600
TotalMilliseconds : 21081600000
```

HINWEIS

Geben Sie nur *sStart* oder nur *-end* an, ersetzt *New-Timespan* die fehlende Angabe mit der aktuellen Zeit. Liegt das Startdatum nach dem Enddatum, sind die Angaben negativ (»Blick in die Vergangenheit«).

Um ein relatives Datum zu ermitteln, verwenden Sie *Get-Date* und ziehen mit *New-Timespan* die gewünschte Zeitspanne ab. Die folgende Zeile ermittelt das Datum vor genau 48 Stunden:

```
PS > ((Get-Date) - (New-TimeSpan -hours 48))
Samstag, 22. Januar 2011 13:41:52
```

Mithilfe relativer Daten finden Sie beispielsweise alle Logdateien im Windows-Ordner, die älter sind als 30 Tage:

```
PS > $stichtag = ( (Get-Date) - (New-TimeSpan -days 30))
PS > Dir $env:windir -filter *.log -ea 0 | Where-Object { $_.LastWriteTime -lt →
    $stichtag }
```

Sie könnten ebenso gut alle *Error*-Ereignisse des System-Ereignislogbuchs der letzten 48 Stunden abrufen.

```
PS > $stichtag = ( (Get-Date) - (New-TimeSpan -hours 48) )
PS > Get-Eventlog System -EntryType Error -After $stichtag
```

Auch andere Cmdlets können *TimeSpan*-Objekte als Ergebnis zurückliefern. Die nächste Zeile ermittelt mithilfe von *Measure-Command*, wie lange ein PowerShell-Befehl für die Ausführung benötigt:

```
PS > (Measure-Command { dir $env:windir }).TotalSeconds
0,0513835
```

Hintergrund

Subtrahieren Sie ein Datum von einem anderen, berechnet PowerShell automatisch die Zeitdifferenz und liefert als Ergebnis ein *TimeSpan*-Objekt, das die Zeitdifferenz in verschiedenen Einheiten zurückgibt. Voraussetzung dafür ist allerdings, dass Sie *DateTime*-Objekte voneinander subtrahieren. Normaler Text liefert auf diese Weise natürlich keine Zeitdifferenz:

```
PS > '1.3.2008' - '4.12.2007'
Fehler beim Aufrufen der Methode, da [System.String] keine Methode mit dem Namen
"op_Subtraction" enthält.
```

Wandeln Sie die beiden Zeitangaben hingegen jeweils in ein *DateTime*-Objekt um, funktioniert die Berechnung einwandfrei:

```
PS > (Get-Date '1.3.2008') - (Get-Date '4.12.2007')

Days           : 88
Hours          : 0
Minutes        : 0
(...)
TotalMinutes   : 126720
TotalSeconds   : 7603200
TotalMilliseconds : 7603200000
```

Weil Zeitunterschiede häufig benötigt werden, erleichtert das Cmdlet *New-TimeSpan* die Zeitdifferenzberechnung. Dieses Cmdlet wandelt die angegebenen Daten automatisch für Sie um:

```
PS > New-TimeSpan '4.12.2007' '1.3.2008'

Days           : 88
Hours          : 0
Minutes        : 0
Seconds        : 0
(...)
TotalMinutes   : 126720
TotalSeconds   : 7603200
TotalMilliseconds : 7603200000
```


Geben Sie nur ein Datum an, wird automatisch mit dem aktuellen Datum verglichen. *New-Timespan* interpretiert Datums- und Zeitangaben genau wie *Get-Date* entsprechend der aktuellen Kultur, erwartet auf deutschen Systemen also ein deutsches Datumsformat.

Das *Timespan*-Objekt besteht aus einer Reihe von Eigenschaften, die die Zeitspanne in verschiedenen Einheiten beschreiben. Möchten Sie auf eine davon zugreifen, verwenden Sie entweder *Select-Object* in Verbindung mit dem Parameter *-expandProperty* oder Sie greifen direkt auf die Punkt-Schreibweise zurück:

```
PS > New-TimeSpan '4.12.2007' '1.3.2008' | Select-Object -expandProperty Days
88
PS > (New-TimeSpan '4.12.2007' '1.3.2008').Days
88
```

Möchten Sie das Alter von Dateien bestimmen, greifen Sie mit *Get-Item* zunächst auf die gewünschte Datei zu und verwenden dann dessen Eigenschaft *CreationTime*. Weil diese Eigenschaft bereits im *DateTime*-Format vorliegt, muss sie nicht mehr umgewandelt werden:

```
PS > "Test" | Out-File $env:temp\testfile.txt
PS > (Get-Item $env:temp\testfile.txt).CreationTime
Montag, 14. Februar 2011 10:38:32
```

Sie dürfen übrigens dieser Dateieigenschaft auch ein neues Datum zuweisen. Die folgende Zeile ändert das Erstelldatum der Datei auf den 1. Mai 1718 um zirka halb acht Uhr abends:

```
PS > (Get-Item $env:temp\testfile.txt).CreationTime = '1.5.1718 19:33:12'
```

Relative Daten, die häufig zur Berechnung von Stichtagen verwendet werden, ermitteln Sie entweder, indem Sie vom aktuellen Datum eine mit *New-Timespan* berechnete Zeitspanne abziehen. Das Datum vor 7 Tagen erhalten Sie also so:

```
PS > ((Get-Date) - (New-Timespan -Days 7))
```

Oder Sie verwenden die .NET-Methode *AddDays()* und ziehen die benötigte Anzahl der Tage direkt vom aktuellen Datum ab:

```
PS > (Get-Date).AddDays(-7)
```

Methode	Beschreibung
<i>Add()</i>	Fügt eine Zeitdifferenz hinzu
<i>AddDays()</i>	Fügt Tage hinzu oder zieht sie ab
<i>AddHours()</i>	Fügt Stunden hinzu oder zieht sie ab

Tabelle 2.5 Methoden eines *DateTime*-Objekts, um Zeitintervalle hinzuzufügen oder abzuziehen

Methode	Beschreibung
<i>AddMilliseconds()</i>	Fügt Millisekunden hinzu oder zieht sie ab
<i>AddMinutes()</i>	Fügt Minuten hinzu oder zieht sie ab
<i>AddMonths()</i>	Fügt Monate hinzu oder zieht sie ab
<i>AddSeconds()</i>	Fügt Sekunden hinzu oder zieht sie ab
<i>AddTicks()</i>	Fügt Systemticks hinzu oder zieht sie ab
<i>AddYears()</i>	Fügt Jahre hinzu oder zieht sie ab

Tabelle 2.5 Methoden eines *DateTime*-Objekts, um Zeitintervalle hinzuzufügen oder abzuziehen (*Fortsetzung*)

WMI-Zeitangaben umwandeln

Sie möchten eine Zeitinformation, die Sie von WMI (Windows Management Instrumentation) erhalten haben, in ein lesbares Datums- und Zeitformat umwandeln.

Lösung

Verwenden Sie die Methode *ConvertToDateTime()*:

```
PS > $bootup = (Get-WmiObject Win32_OperatingSystem).LastBootUpTime
PS > $bootup
20110413151635.764544+120

PS > $datum = ([wmi]'').ConvertToDateTime($bootup)
PS > $datum
Sonntag, 13. April 2011 15:16:35

PS > ((Get-Date) - $datum).TotalHours
69,9004367377778
PS > 'Ihr System läuft seit {0:0.0} Stunden.' -f ((New-Timespan $datum).TotalHours)
Ihr System läuft seit 69,9 Stunden.
```

Hintergrund

WMI verwaltet Zeitinformationen in einem eigenen universellen und international einheitlichen Format. Dieses Format besteht aus der Datums-, der Zeit- und der Zeitzoneninformation. WMI stellt mit *ConvertToDateTime()* eine Methode zur Verfügung, um diese WMI-spezifische Zeitinformation in ein Standard-*DateTime*-Objekt zu verwandeln. Damit können Sie die üblichen Zeitberechnungen durchführen.

»[wmi]« ist einsogenannter *Type Accelerator*, also eine Abkürzung, für die Klasse *System.Management.ManagementObject*.

```
PS > ([wmi]'').GetType().FullName  
System.Management.ManagementObject
```

Sie könnten die Konversion auch folgendermaßen durchführen:

```
PS > $objekt = New-Object System.Management.ManagementObject  
PS > $objekt.ConvertToDateTime($bootup)  
Sonntag, 13. April 2008 15:16:35
```

Systemticks umwandeln

Sie möchten eine Zeitangabe, die in Systemticks (Zeittakten) angegeben ist, in ein lesbares Datum umwandeln.

Lösung

Systemticks sind die kleinste Zeiteinheit eines Computers. Sie können ein beliebiges Datum mit der Eigenschaft *Ticks* in Systemticks ausgeben:

```
PS > (Get-Date).Ticks  
633483668733250000
```

Auf diese Weise lassen sich auch Zeitabstände messen:

```
PS > $snap1 = Get-Date  
PS > dir  
(...)  
PS > $snap2 = Get-Date  
PS > (New-TimeSpan $snap1 $snap2).Ticks  
12948000
```

Möchten Sie Ticks in eine lesbare Zeit umwandeln, verwenden Sie die statischen Methoden *FromFileTime()* oder *FromFileTimeUTC()* der *DateTime*-Klasse:

```
PS > $ticks = (New-TimeSpan $snap1 $snap2).Ticks  
PS > $datum = [DateTime]::FromFileTimeUTC($ticks)  
PS > $datum  
Montag, 1. Januar 1601 00:00:01  
  
PS > $sekunden = $datum.ToString('s')  
PS > $millisekunden = $datum.ToString('fff')  
PS > "Ausführungszeit $sekunden sec und $millisekunden ms."  
Ausführungszeit 1 sec und 294 ms.
```

Hintergrund

Ticks werden vom Beginn des Gregorianischen Kalenders an gezählt. Dieser historische Zeitpunkt ist auf den 1.1.1601 festgelegt. Deshalb zeigt das Datum den 1. Januar 1601 an, weil in diesem Beispiel die Ticks eine relative Zeit (nämlich die Zeitdifferenz aus zwei Schnappschüssen) darstellen. Absolute Daten lassen sich aber ebenso umwandeln:

```
PS > $ticks = (Get-Date).Ticks
PS > $ticks
633483678960694000
PS > [DateTime]::FromFileTimeUTC($ticks)
Freitag, 6. Juni 3608 16:51:36
```

Analog lassen sich Ticks auch umwandeln, indem Sie sie mit *AddTicks()* zum 1.1.1601 hinzufügen:

```
PS > $ticks = (Get-Date).Ticks
PS > $ticks
633483679545850000
PS > $datum = Get-Date 1.1.1601
PS > $datum.AddTicks($ticks)
Freitag, 6. Juni 3608 16:52:34
```

Zusammenfassung

In diesem Kapitel haben Sie diese PowerShell-Cmdlets kennengelernt:

Name	Beschreibung
<i>Get-Date</i>	Liefert das aktuelle Datum und die aktuelle Zeit Liefert ein beliebiges Datum oder eine beliebige Zeit als <i>DateTime</i> -Objekt Wandelt ein Datum in Systemticks um
<i>Get-Item</i>	Greift auf eine Datei zu, um beispielsweise Datumsinformationen wie das Erstelldatum auszulesen
<i>Get-WmiObject</i>	Liefert Informationen zu Computereigenschaften wie beispielsweise die Zeit des letzten Systemstarts
<i>New-TimeSpan</i>	Berechnet die Zeitdifferenz zwischen zwei Zeitinformationen
<i>Read-Host</i>	Liest Benutzereingaben von der Konsole ein

Tabelle 2.6 Cmdlets aus diesem Kapitel

Darüber hinaus haben Sie mit den folgenden .NET-Klassen und -Objekten gearbeitet:

Name	Beschreibung
<i>[DateTime]</i>	Kurzform für <i>[System.DateTime]</i> ; bietet statische Methoden für den Umgang mit Datum und Zeit. Wandelt Text in ein <i>DateTime</i> -Objekt um.
<i>[System.Globalization.CultureInfo]</i>	Liefert Methoden, mit denen sämtliche verfügbaren Kulturen aufgelistet werden können
<i>[wmi]</i>	Kurzform für <i>[System.Management.ManagementObject]</i> ; bietet Methoden, mit denen das WMI-interne Datums- und Zeitformat in lesbare Angaben umgewandelt werden kann
<i>DateTime</i>	Repräsentiert eine Zeitinformation
<i>System.Globalization.CultureInfo</i>	Liefert Formatierungsinformationen zu einer bestimmten Kultur und legt fest, wie Daten und Zeitangaben kulturentsprechend formatiert werden
<i>TimeSpan</i>	Repräsentiert eine Zeitdifferenz zwischen zwei Zeitinformationen

Tabelle 2.7 .NET-Klassen und -Objekte aus diesem Kapitel

Kapitel 3

Listen und Arrays

In diesem Kapitel:

Neues Array anlegen	96
Mehrdimensionale Arrays verwenden	100
Auf Arrayelemente zugreifen	101
Alle Arrayelemente der Reihe nach bearbeiten	103
Arrayinhalte sortieren	106
Prüfen, ob ein Array ein bestimmtes Element enthält	107
Arrayelemente nachträglich hinzufügen und entfernen	109
Mehrere Arrays zusammenfassen	112
Arrays miteinander vergleichen	113
Schlüssel-Wert-Paare speichern	115
Schlüssel-Wert-Paare sortieren	117
Zusammenfassung	119

Arrays (Variablenfelder) spielen bei PowerShell eine elementare Rolle, weil sie sehr häufig eingesetzt werden. Liefert ein Befehl mehr als ein Ergebnis zurück, verpackt PowerShell diese Ergebnisse in einem Array.

In diesem Kapitel erfahren Sie, wie Sie mit den Inhalten eines Arrays umgehen und zum Beispiel auf einzelne Elemente darin zugreifen oder alle Inhalte eines Arrays der Reihe nach auswerten.

Neues Array anlegen

Sie wollen mehrere Werte in einer einzelnen Variablen speichern.

Lösung

Geben Sie die Inhalte, die Sie in der Variablen speichern wollen, kommasepariert an. Dadurch entsteht ein Array, das dann in der Variablen gespeichert wird:

```
PS > $array = 100,200,5.789,"Text",$true
```

Soll das Array nur ein Element enthalten, formulieren Sie so:

```
PS > $array = , 'Hallo'
PS > $array.Count
1
```

Ein leeres Array erhalten Sie mit der Konstruktion `@()`. Das Array kann dann später nach und nach gefüllt werden:

```
PS > $array = @()
PS > $array += 'Hallo'
PS > $array += 1
PS > $array
Hallo
1
```

Befehle, die mehr als ein Element zurückliefern, verpacken die Ergebnisse automatisch als Array:

```
PS > $array = Get-Process
PS > $array.GetType().FullName
System.Object[]
PS > $array.GetType().isArray
True
```

Um sicherzustellen, dass ein Befehl immer ein Array liefert, also auch dann, wenn er kein oder nur ein Ergebnis liefert, verwenden Sie erneut die Konstruktion `@()`:


```
PS > $result = @(Get-Process notepad -ErrorAction SilentlyContinue)
PS > 'Derzeit laufen {0} Instanzen von Notepad.' -f $result.Count
PS > If ($result.Count -gt 0) { 'Die Prozess-IDs lauten {0}.' -f (($result | Select-Object
-expand ID) -join ', ')} }
```

Hintergrund

Arrays lassen sich manuell auf verschiedene Arten erzeugen, wobei das Komma der gebräuchlichste Weg ist. Sorgen Sie nur dafür, dass die durch Komma separierten Werte eigenständige Daten sind.

Der folgende Aufruf ist zum Beispiel fehlerhaft, weil Sie Datenwerte mit Befehlsaufrufen (*Get-Date*) mischen:

```
PS > $array = 1,2,Get-Date, "Hallo"
Fehlender Ausdruck nach ",".
Bei Zeile:1 Zeichen:14
+ $array = 1,2, <<<< Get-Date, "Hallo"
```

Setzen Sie den Befehlsaufruf in runde Klammern, um ihn als Unterausdruck abzugrenzen. PowerShell wertet dadurch den Befehl zuerst aus und fügt das Ergebnis des Befehls ins Array ein:

```
PS > $array = 1,2,(Get-Date), "Hallo"
PS > $array
1
2

Mittwoch, 16. April 2008 08:53:31
Hallo
```

Eine besondere Form des Arrays sind Zahlenbereiche, die Sie mit dem Operator »..<« anlegen.

```
PS > $array = 1..5
PS > $array
1
2
3
4
5
PS > $array = 5..1
PS > $array
5
4
3
2
1
```

Leider kann »..<« nur Zahlenbereiche erzeugen. Buchstabenfolgen werden dagegen nicht unterstützt. Indem Sie aber ASCII-Codes erzeugen, lassen sich daraus durch Typumwandlung Zeichenfolgen generieren:

```
PS > -join [char[]](65..90)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Die Konstruktion `@()` wandelt die Daten, die Sie innerhalb der runden Klammern angeben, automatisch in ein Array – es sei denn, es handelt sich dabei bereits um ein Array. Deshalb ist diese Konstruktion außerordentlich wichtig, um Befehlsergebnisse in PowerShell-Skripts zu »normalisieren«. Andernfalls wüssten Sie nämlich zur Laufzeit nicht verlässlich, welche Art von Datentypen Sie vom Befehl zurückerhalten. Schließlich können Befehle grundsätzlich drei verschiedene Datentypen zurückliefern:

- **Nullwert** Der Befehl konnte keine Ergebnisse finden
- **Beliebiger Datentyp** Der Befehl konnte genau ein Ergebnis finden und liefert es in einem beliebigen Datentyp zurück. Prozesse werden beispielsweise als *Process*-Objekt zurückgeliefert, während Dateien in Form von *FileInfo*-Objekten zurückgegeben werden.
- **Array** Der Befehl liefert mehr als ein Ergebnis. Deshalb verpackt PowerShell die Einzelergebnisse in einem Array.

Indem Sie den Befehl in die Konstruktion `@()` stellen, sorgen Sie dafür, dass immer ein Array zurückgeliefert wird. Konnte der Befehl keine Ergebnisse liefern, ist es leer. Andernfalls enthält es die Ergebnisse, und zwar auch dann, wenn nur ein Ergebnis vom Befehl ermittelt wurde.

Die in PowerShell üblichen Arrays werden auch »Jagged Arrays« genannt, denn sie unterscheiden sich von den klassischen symmetrischen Arrays. Jedes Arrayelement kann in sich wiederum ein Array sein, wodurch sich pro Element unterschiedliche Dimensionen ergeben. Diese Form der Arraygestaltung ist sehr speichereffizient, weil nur so viele Arrayelemente angelegt werden, wie auch tatsächlich benötigt werden:

```
PS > $array = 1,(1..10),5
PS > $array[1][4] = 'Hello', 'World'
PS > $array[1][4][0]
Hello
```

TIPP

Benötigen Sie ein Array von Strings, kann `@()` ebenfalls sehr nützlich sein. Schauen Sie sich die Ergebnisse der folgenden beiden Zeilen an:

```
PS > 'localhost' * 10
PS > @('localhost') * 10
```

Arrays sind normalerweise vom Typ *Object*, was bedeutet, dass die einzelnen Arrayelemente einen beliebigen Datentyp besitzen dürfen. Das ist der Grund, warum Sie in Arrays beispielsweise Zahlen und Text miteinander mischen dürfen.

Möchten Sie ein streng typisiertes Array erstellen, das nur den von Ihnen vorgegebenen Datentyp speichert, erstellen Sie eine Variable, der Sie den gewünschten Datentyp voranstellen. Da es sich nicht um einen Einzelwert, sondern um ein Array handeln soll, fügen Sie eine geöffnete und eine geschlossene eckige Klammer hinter dem Datentypnamen an. Danach speichern Sie das Array in dieser Variable. Das folgende Beispiel legt eine Variable an, die nur Arrays speichert, die aus 32-Bit-Ganzzahlen (Typ *Int32*) bestehen:

```
PS > [Int32[]]$array = 1,2.4,3.9,4.8,5
PS > $array += 10.1
PS > $array
1
2
4
5
5
10
```

TIPP

Der .NET-Typ *Array* enthält nützliche statische Methoden für Arrays. Mit *Reverse()* können Sie beispielsweise den Inhalt eines Arrays »umdrehen«:

```
PS > $array = 1..4
PS > [Array]::Reverse($array)
PS > $array
4
3
2
1
```

Mit *Sort()* lassen sich Arrayinhalte sortieren:

```
PS > $array = 1,4,2,7,6,3
PS > [Array]::Sort($array)
1
2
3
...
```

Indem Sie ein Array vorübergehend in den Typ *ArrayList* umwandeln, erhalten Sie weitere sehr nützliche Arraymethoden wie *RemoveAt()*, mit der Sie einzelne Elemente aus einer bestimmten Position des Felds entfernen können, oder *Add()*, um neue Elemente an das Array anzuhängen, ohne dass dabei wie beim Operator »+=« das gesamte Array zeitintensiv neu angelegt werden muss:

```
$array = 1..10
$betterarray = [System.Collections.ArrayList]$array
$betterarray.RemoveAt(4)
$betterarray
$array = $betterarray.ToArray()
$array
```

Alle Eigenschaften und Methoden des Typs *ArrayList* erhalten Sie mit *Get-Member*:

```
PS > $betterarray | Get-Member
```

Mehrdimensionale Arrays verwenden

Sie möchten ein mehrdimensionales Array mit mehreren Indizes verwenden, zum Beispiel, um ein zwei- oder dreidimensionales Spielfeld abzubilden.

Lösung

Legen Sie ein mehrdimensionales Array mit *New-Object* an. Die folgende Zeile legt ein Array mit zehn Elementen in der ersten und zwanzig Elementen in der zweiten Dimension an, was zum Beispiel einem 10 Arrays breiten und 20 Arrays langen Spielarray mit 200 Arrays entspricht.

```
PS > $array = New-Object "object[,] " 10,20
```

Sie können nun die Arrayelemente über die beiden Indizes ansprechen. Das erste Element trägt zum Beispiel den Index 0,0:

```
PS > $array[0,0]="Erstes Array"
```

Das letzte Array trägt den Index 9,19:

```
PS > $array[9,19]="Letztes Array"
```

Insgesamt enthält das Array Platz für 200 Elemente ($10 * 20 = 200$):

```
PS > $array.length  
200
```

Hintergrund

Klassische multidimensionale Arrays bilden Gitterstrukturen ab. Sie legen dabei für jede Dimension fest, wie viele Elemente darin gespeichert werden können. Deshalb sind klassische multidimensionale Arrays speicherintensiv. PowerShell unterstützt solche Arrays nur indirekt, indem Sie mit *New-Object* das gewünschte mehrdimensionale Array aus .NET Framework abrufen. Normalerweise verwendet PowerShell irreguläre Arrays, die sogenannten »Jagged Arrays«, die Sie bereits kennengelernt haben.

Der Unterschied zwischen echten multidimensionalen Arrays und zusammengesetzten eindimensionalen Arrays wird bereits bei der Schreibweise deutlich. Bei echten multidimensionalen Arrays wird der mehrdimensionale Index innerhalb der eckigen Klammern angegeben. Bei

zusammengesetzten eindimensionalen Arrays rufen Sie zuerst mit einer Kombination aus eckigen Klammern das Arrayelement des eindimensionalen Arrays ab. Ist es seinerseits ein weiteres eindimensionales Array, geben Sie eine zweite eckige Klammer an, um auf dessen gewünschtes Element zuzugreifen.

Auf Arrayelemente zugreifen

Sie verfügen über ein Array und möchten einzelne oder alle Elemente in diesem Array anzeigen.

Lösung

Möchten Sie den gesamten Inhalt eines Arrays sichtbar machen, geben Sie es einfach aus. PowerShell wandelt dann die Arrayelemente in Text um und schreibt sie untereinander:

```
PS > $array = 100,200,5.789,"Text",$true
PS > $array
100
200
5,789
Text
True
```

Auf einzelne Arrayelemente greifen Sie über Indexzahlen in eckigen Klammern zu. Das erste Element erreichen Sie stets über den Index 0, das zweite über den Index 1 etc. Negative Zahlen zählen von hinten, sodass der Index -1 das letzte Element liefert. Geben Sie mehrere Indizes an, werden mehrere Elemente gleichzeitig aus dem Array gelesen:

```
PS > $array[0]
100
PS > $array[-1]
True
PS > $array[0,-1]
100
True
```

Die Anzahl von Elementen in einem Array verrät die Eigenschaft *Count* des Arrays, sodass die folgende Anweisung die Anzahl der Elemente im Array liefert:

```
PS > $array.Count
5
```

Liefert ein Befehl mehr als ein Ergebnis zurück, ist das Ergebnis ebenfalls ein Array. Die folgende Zeile liefert die drei Prozesse mit der höchsten CPU-Belastung:

```
PS > (Get-Process | Sort-Object CPU -Descending)[0..2]
```

Diese Zeile liefert die fünf größten Dateien im Windows-Ordner:

```
PS > (Dir $env:windir | Sort-Object Length -Descending) [0..4]
```

ACHTUNG Greifen Sie auf ein Arrayelement zu, das gar nicht im Array enthalten ist, erhalten Sie keine Fehlermeldung, sondern einfach nur nichts zurück:

```
PS > $array = 1..10
PS > $array[5]
6
PS > $array[100]
```

Hintergrund

PowerShell erleichtert den Umgang mit Arrayelementen, weil es den Inhalt eines Arrays automatisch sichtbar macht, wenn Sie das Array ausgeben. Dabei verwendet PowerShell normalerweise den Zeilenumbruch als Trennzeichen, sodass die Elemente untereinander ausgegeben werden.

Möchten Sie ein anderes Trennzeichen verwenden, dann beauftragen Sie PowerShell, das Array explizit in Text umzuwandeln, indem Sie die Variable, die das Array speichert, mithilfe von doppelten Anführungszeichen auflösen lassen. Jetzt verwendet PowerShell als Trennzeichen ein Leerzeichen, und mit der besonderen Variable *\$OFS* («Object Field Separator») können Sie auch andere Trennzeichen festlegen:

```
PS > $ofs = " + "
PS > $array
100
200
5,789
Text
True
PS > "$array"
100 + 200 + 5.789 + Text + True
```

Noch flexibler funktioniert der Operator *-join*, weil Sie hier weder die explizite Textumwandlung benötigen noch das Trennzeichen in einer besonderen Variable zu hinterlegen brauchen. Stattdessen geben Sie links vom Operator lediglich das Array und rechts vom Operator das gewünschte Trennzeichen an:

```
PS > $array -join ' -<-> '
```

Da die meisten Befehle Arrays zurückliefern, wenn es mehr als ein Ergebnis gibt, lassen sich daraus mithilfe von *-join* leicht kommaseparierte Listen erstellen. Dieses Beispiel liest die Namen aller laufenden Prozesse und gibt sie als kommaseparierte sortierte Liste aus:

```
PS > (Get-Process | Select-Object -expandProperty Name | Sort-Object) -join ', '
AcroRd32, AdobeARM, alg, AppleMobileDeviceService, ApplicationUpdater, conhost, conhost,
csrss, csrss, DataCardMonitor, dllhost, dwm, ehrecvr, esClient, explorer, fdm,
FlashUtil10k ActiveX, Idle, iexplore, iexplore, iexplore, iPodService, iTunesHelper,
LightsOutClientService, lsass, lsm, mDNSResponder, Microsof...
```

Hier wird der Hersteller der laufenden Prozesse erfragt und daraus eine semikolonseparierte sortierte Liste der Softwareanbieter erstellt:

```
PS > (Get-Process | ForEach-Object { $_.Company } | Sort-Object -Unique | 
    Where-Object { $_ }) -join '; '
Adobe Systems Incorporated; Adobe Systems, Inc.; Apple Inc.; Deutsche Post AG; Huawei
Technologies Co., Ltd.; Idera; Microsoft Corporation; PowerISO Computing, Inc.
```

Alle Arrayelemente der Reihe nach bearbeiten

Sie möchten alle Elemente eines Arrays der Reihe nach weiterverarbeiten.

Lösung

Verwenden Sie eine *Foreach*-Schleife, um alle Arrayelemente der Reihe nach aus dem Array zu lesen und weiterzubearbeiten.

```
PS > $array = 1..5
```

Setzen Sie dazu entweder die Pipeline ein:

```
PS > $array | ForEach-Object { "Bearbeite Arrayelement $_" }
Bearbeite Arrayelement 1
Bearbeite Arrayelement 2
Bearbeite Arrayelement 3
Bearbeite Arrayelement 4
Bearbeite Arrayelement 5
```

Oder verwenden Sie einen *Foreach*-Block:

```
PS > Foreach ($element in $array) { "Bearbeite Arrayelement $element" }
Bearbeite Arrayelement 1
Bearbeite Arrayelement 2
Bearbeite Arrayelement 3
Bearbeite Arrayelement 4
Bearbeite Arrayelement 5
```

Sie können auch eine *For*-Schleife verwenden und die Arrayelementindizes verwenden:

```
PS > For ($x=0; $x -lt $array.length; $x++) { $x }
0
1
2
3
4
PS > For ($x=0; $x -lt $array.length; $x++) { "Element $x = $($array[$x])" }
Element 0 = 1
Element 1 = 2
Element 2 = 3
Element 3 = 4
Element 4 = 5
```

Möchten Sie die Arrayelemente rückwärts aufzählen, zählen Sie vom höchsten zum niedrigsten Element:

```
PS > For ($x=$array.length-1; $x -ge 0; $x--) { $x }
4
3
2
1
0
PS > For ($x=$array.length-1; $x -ge 0; $x--) { "Element $x = $($array[$x])" }
Element 4 = 5
Element 3 = 4
Element 2 = 3
Element 1 = 2
Element 0 = 1
```

Hintergrund

Da jedes Arrayelement über eine eindeutige Indexzahl verfügt und da alle Arrayindizes bei 0 beginnen, benötigen Sie nur noch die Gesamtzahl der Elemente im Array, um alle Elemente in einer *For*-Schleife anzusprechen. Zählen Sie also entweder aufwärts von 0 bis zur Anzahl der Arrayelemente minus 1 oder abwärts von der Anzahl der Arrayelemente minus 1 bis 0. Die *For*-Schleife ist die schnellste Schleifenkonstruktion, aber häufig etwas kryptisch und schlechter nachvollziehbar als die übrigen Schleifen.

Ein übersichtlicherer Weg ist die *Foreach*-Schleife, denn sie liest der Reihe nach alle Elemente eines Arrays, ohne dass Sie sich um die Gesamtzahl oder Indexzahlen der einzelnen Elemente kümmern müssen. Mit *Foreach* kann man deshalb auch Arrayelemente von Arrays aufzählen, die nicht über einen simplen zahlenbasierten Index verfügen. Solche Arrays erhalten Sie häufig als Ergebnis von externen Befehlsaufrufen wie WMI oder COM. Die *Foreach*-Schleife steht in zwei Formen zur Verfügung: als Pipelinebefehl (*ForEach-Object*) und als klassische *Foreach*-Schleifenkonstruktion.

```
PS > $array = 1,2,3,4
PS > $array | ForEach-Object { "bearbeite gerade $_" }
PS > Foreach($Element in $array) { "bearbeite gerade $Element" }
```


Während *ForEach-Object* sehr speicherplatzeffizient ist, weil in der Pipeline stets nur ein Element im Speicher gehalten zu werden braucht, ist die *ForEach*-Konstruktion in der Regel rund zwanzig Mal schneller: Sie verzichtet auf den relativ aufwändigen Übergabemechanismus der Pipeline.

Geht es Ihnen nur darum, die Inhalte eines Arrays aufzusummieren oder andere statistische Berechnungen durchzuführen, verwenden Sie *Measure-Object*.

```
PS > $array = 1..100
PS > $array | Measure-Object -sum -average -maximum -minimum

Count      : 100
Average    : 50,5
Sum        : 5050
Maximum    : 100
Minimum    : 1
Property   :
```

Das setzt voraus, dass das Array nur Zahlenwerte enthält. Speichert Ihr Array dagegen komplexe Objekte, wählen Sie mit dem Parameter *-property* die Eigenschaft der Objekte aus, die als Berechnungsgrundlage dienen soll. Diese Eigenschaft muss einen numerischen Inhalt haben:

```
PS > $array = dir $env:windir
PS > $array | Measure-Object -sum -average -maximum -minimum -property Length

Count      : 182
Average    : 1186770,88461538
Sum        : 215992301
Maximum    : 95589304
Minimum    : 0
Property   : Length
```

Möchten Sie die Größenverteilung von Dateien in einem Ordner analysieren, beschränken Sie das Ergebnis mit einem Filter auf Dateien, bevor Sie es an *Measure-Object* weiterleiten. Speichern Sie das Ergebnis von *Measure-Object* in einer Variablen, um auf die einzelnen errechneten Werte zugreifen zu können:

```
PS > $ergebnis = $array | Measure-Object -sum -maximum -minimum -average -property Length
PS > '{0} Dateien sind im Schnitt {1:0.0} KB groß. Die größte Datei ist {2:0.0} MB groß. Insgesamt sind {3:0.0} MB belegt' -f $ergebnis.Count, ($ergebnis.Average / 1KB), ($ergebnis.Maximum / 1MB), ($ergebnis.Sum / 1MB)
182 Dateien sind im Schnitt 1159,0 KB groß. Die größte Datei ist 91,2 MB groß. Insgesamt sind 206,0 MB belegt
```

Arrayinhalte sortieren

Sie möchten den Inhalt in einem Array nachträglich nach einem bestimmten Kriterium sortieren.

Lösung

Verwenden Sie *Sort-Object*. Enthält das Element Zahlen oder Text, geben Sie den Inhalt über die Pipeline an *Sort-Object* weiter:

```
PS > $array = 1..10
PS > $array | Sort-Object -descending
10
9
8
7
...
```

Enthält das Array komplexe Objekte, geben Sie die Eigenschaft des Objekts an, nach der Sie sortieren wollen:

```
PS > $array = dir
PS > $array | Sort-Object Length
```

Möchten Sie mehrere Eigenschaften sortieren, geben Sie die Eigenschaften kommasepariert an. Die folgende Zeile sortiert zuerst alphabetisch nach der Dateierweiterung. Gibt es mehrere gleiche Dateierweiterungen, wird anschließend numerisch nach Dateigröße sortiert. Sortiert wird diesmal aufgrund des Parameters *-descending* in absteigender Reihenfolge:

```
PS > $array = dir
PS > $array | Sort-Object Extension, Length -descending
```

Hintergrund

Sort-Object sortiert beliebige Objekte, die es über die Pipeline empfängt, also auch Arrayinhalte. Dabei sortiert *Sort-Object* als Vorgabe in aufsteigender Reihenfolge. Möchten Sie lieber in absteigender Reihenfolge sortieren, geben Sie den Parameter *-descending* an.

Geben Sie keine besondere Eigenschaft an, nach der sortiert werden soll, sortiert *Sort-Object* die Textrepräsentation des Objekts. Auf die Angabe einer Sortiereigenschaft sollten Sie deshalb nur dann verzichten, wenn es sich um ein simples Objekt wie Zahlen oder Text handelt.

Möchten Sie nach mehreren Kriterien sortieren, geben Sie die Kriterien als Array an, also kommasepariert. Allerdings sortiert *Sort-Object* dann alle angegebenen Eigenschaften einheitlich auf- oder absteigend. Wollen Sie für jedes Kriterium separat festlegen, ob es auf- oder absteigend sortiert wird, verwenden Sie ein Hashtable, um diese zusätzlichen Angaben festzulegen.

Die folgenden Zeilen geben das Ordnerlisting folgendermaßen aus:

- Aufsteigend sortiert, zuerst nach Dateityp, dann nach Dateigröße
- Absteigend sortiert nach Dateityp, aufsteigend sortiert nach Dateigröße
- Aufsteigend sortiert nach Dateityp, absteigend sortiert nach Dateigröße

```
PS > $array = dir
PS > $array | Sort-Object extension, length -descending
PS > $array | Sort-Object @{e='extension';a=$false}, @{e='length';a=$true}
PS > $array | Sort-Object @{e='extension';a=$true}, @{e='length';a=$false}
```

Mithilfe von *Sort-Object* können Sie auch Arrayinhalte permanent sortieren, indem Sie das Ergebnis der Sortierung erneut dem Array zuweisen:

```
PS > $array = 1..10
PS > $array
1
2
3
4
(...)
PS > $array = $array | Sort-Object -desc
PS > $array
10
9
8
7
(...)
```

Möchten Sie den Inhalt des Arrays sortieren, weisen Sie es der ursprünglichen oder einer neuen Variable zu. Um ein Array zu sortieren, können Sie auch auf .NET Framework zugreifen und die schnellere Methode *Sort()* des Typs *Array* einsetzen:

```
PS > [Array]::Sort($array)
```

Prüfen, ob ein Array ein bestimmtes Element enthält

Sie möchten feststellen, ob ein Array ein bestimmtes Element enthält, oder Sie möchten bestimmte Elemente aus einem Array lesen.

Lösung

Mit dem Operator *-contains* prüfen Sie, ob ein Array ein bestimmtes Element enthält:

```
PS > $array = 1..10
PS > $array -contains 3
True
PS > $array -contains 100
False
```

Das folgende Beispiel prüft, ob der aktuelle Wochentag in einer Liste vorgegebener Wochentage vorkommt, und führt nur dann eine Aufgabe durch:

```
PS > if ('Monday', 'Friday' -contains (Get-Date).DayOfWeek) {
>> 'führe Aufgabe durch'
>> } else {
>> 'keine Ausführung'
>> }
>>
```

Das Ergebnis von `-contains` ist also stets `$true` oder `$false` und kann die Grundlage für Bedingungen sein.

Möchten Sie dagegen Elemente aus dem Array anzeigen, die Ihrem Suchbegriff entsprechen, verwenden Sie den Vergleichsoperator `-eq`:

```
PS > $array = 1..200
PS > $array -eq 3
3
PS > $array -eq 100
```

Hintergrund

Der Operator `-contains` meldet `$true` zurück, wenn mindestens ein Arrayelement den angegebenen Wert enthält. Der Operator `-eq` dagegen liefert sämtliche Arrayelemente zurück, die dem Suchbegriff entsprechen. Diese Operatoren können auf jedes beliebige eindimensionale Array angewendet werden.

```
PS > $array = 1..10 + 1..10
PS > $array -contains 1
True
PS > $array -eq 1
1
1
```

Wollen Sie zum Beispiel ermitteln, wie häufig eine bestimmte Zahl in einer Zahlenliste vorkommt, suchen Sie zuerst mit `-eq` danach und zählen dann das Ergebnis. Die nächsten Zeilen zeigen, wie das funktioniert, und ermitteln, dass in der Zahlenliste die Zahl 2 genau dreimal vorkommt:

```
PS > $array = 1,2,6,43,2,2,5,7,22,4,7,3
PS > @($array -eq 2).Count
3
```

Verwenden Sie anstelle von `-eq` den Operator `-like`, dann dürfen Sie Platzhalterzeichen verwenden. Die folgende Zeile würde alle Arrayelemente finden, die die Zahl »2« enthalten, also auch »22«:

```
PS > $array -like '*2*'
```

Auch die übrigen gebräuchlichen mathematischen Operatoren funktionieren mit Arrays. So würde die nächste Anweisung alle Arrayelemente liefern, die größer oder gleich 10 sind:

```
PS > $array -ge 10
```

Mit dem Operator *-match* dürfen schließlich auch komplexe reguläre Ausdrücke verwendet werden. Die nächsten Zeilen listen alle Arrayelemente auf, die die Zahlen 1, 3 oder 6 enthalten:

```
PS > $array = 1..100
PS > $array -match '[136]'
```

1
3
6
10
11
12
...

TIPP

Mehr Kontrolle liefert *Where-Object*, denn mit diesem Filter-Cmdlet können Sie einen Skriptblock definieren, der entscheidet, ob ein Element aus der Pipeline gefiltert wird oder nicht. Das folgende Beispiel zeigt nur die Arrayelemente an, deren Länge größer ist als vier Zeichen:

```
PS > 'Hans','Werner','Tobias','Mary' | Where-Object { $_.Length -gt 4 }
```

Arrayelemente nachträglich hinzufügen und entfernen

Sie möchten Elemente aus einem Array entfernen oder neue Elemente hinzufügen.

Lösung

Neue Elemente fügen Sie am Ende eines vorhandenen Arrays hinzu, indem Sie den »+=«-Operator verwenden:

```
PS > $array = 1..4
PS > $array += 5
PS > $array
```

1
2
3
4
5

Möchten Sie ein bestimmtes Arrayelement entfernen, fügen Sie das Array aus den übrigen Elementen neu zusammen. Um beispielsweise das Element mit dem Index 4 zu entfernen, gehen Sie so vor:

```
PS > $array = $array[0..3] + $array[5..($array.length-1)]
PS > $array
```

Hintergrund

Es ist technisch eigentlich gar nicht möglich, aus Arrays nachträglich Elemente herauszustreichen oder neue Elemente hinzuzufügen. PowerShell muss also bei solchen Änderungen jeweils komplett neue Arrays anlegen, was einen hohen Rechenaufwand bedeutet.

Bei kleineren Arrays spielt der zwar keine Rolle, aber wenn Ihre Aufgabe große Arrayinhalte betrifft oder Sie diese Änderungen sehr häufig durchführen möchten, sind die normalen Arrays dafür nicht gut geeignet.

In solch einem Fall speichern Sie Ihre Informationen besser in einem .NET-Objekt namens *ArrayList*. Dieses Objekt verfügt über zusätzliche Methoden wie *Add()* und *Remove()*, mit denen Sie Elemente effizient einfügen und auch wieder entfernen können. Die folgenden Zeilen legen eine neue *ArrayList* an und speichern darin mit *Add()* einige Werte:

```
PS > $array = New-Object System.Collections.ArrayList
PS > $array.Count
0
PS > $array.Add('Ein neues Element')
0
PS > $array.Add( (Get-Date) )
1
PS > $array.Add(100)
2
PS > $array.Count
3
PS > $array
Ein neues Element
Mittwoch, 13. April 2011 10:01:04
100
```

HINWEIS

Die Methode *Add()* fügt das angegebene Element nicht nur in die *ArrayList* ein, sondern gibt die Listenposition zurück, an der das Element gespeichert wurde. Interessiert Sie diese Angabe nicht, sollten Sie sie entsorgen, indem Sie den Rückgabewert von *Add()* in den Datentyp *Void* umwandeln:

```
PS > $array.Add(100)
3
PS > [void]$array.Add(100)
```

Während *Add()* die neuen Elemente stets am Ende der Liste einfügt, dürfen Sie bei *Insert()* die Listenposition frei wählen. Die folgende Zeile fügt das Element zum Beispiel am Listenanfang ein (Indexposition 0):

```
PS > $array.Insert(0, 'Ganz am Anfang')
```

Arrayelemente lassen sich mit *Remove()* auch wieder entfernen, indem Sie den Inhalt des Elements angeben. Existiert das angegebene Element mehrfach, wird es nur einmal entfernt:

```
PS > $array.Remove('Ein neues Element')
```

```
PS > $array
```

```
Ganz am Anfang
```

```
Mittwoch, 13. April 2011 10:01:04
```

```
100
```

```
100
```

```
100
```

```
PS > $array.Add(100)
```

```
5
```

```
PS > $array.Add(100)
```

```
6
```

```
PS > $array
```

```
Ganz am Anfang
```

```
Mittwoch, 13. April 2011 10:01:04
```

```
100
```

```
100
```

```
100
```

```
100
```

```
100
```

```
PS > $array.Remove(100)
```

```
PS > $array
```

```
Ganz am Anfang
```

```
Mittwoch, 13. April 2011 10:01:04
```

```
100
```

```
100
```

```
100
```

```
100
```

```
PS > $array.Remove(100)
```

```
PS > $array
```

```
Ganz am Anfang
```

```
Mittwoch, 13. April 2011 10:01:04
```

```
100
```

```
100
```

```
100
```

Mit *RemoveAt()* entfernen Sie Arrayelemente an einer bestimmten Position innerhalb der *Array-List*:

```
PS > $array.RemoveAt(0)
PS > $array
Mittwoch, 13. April 2011 10:01:04
100
100
100
100
100
```

Sie können ein Arrayelement übrigens auch entfernen, indem Sie den Operator *-ne* (ungleich) verwenden. Die folgende Zeile entfernt sämtliche Elemente des Arrays, die dem angegebenen Element entsprechen:

```
PS > $array = $array -ne 100
PS > $array
Mittwoch, 13. April 2011 10:01:04
```

Mehrere Arrays zusammenfassen

Sie möchten mehrere einzelne Arrays zu einem Array zusammenfassen.

Lösung

Addieren Sie die Arrays. PowerShell legt daraufhin ein neues Array an, das sämtliche Elemente der addierten Arrays enthält:

```
PS > $logs = dir $env:windir\*.log -name
PS > $exes = dir $env:windir\*.exe -name
PS > $alle = $logs + $exes
PS > $alle
```

Hintergrund

Zwar kann PowerShell klassische eindimensionale Arrays nicht erweitern oder verkleinern. Wenn Sie jedoch den »+«-Operator auf solche Arrays anwenden, erzeugt PowerShell automatisch ein neues Array und kopiert sämtliche Inhalte der angegebenen Arrays in das neue Array.

Weil *\$array3* dabei ein komplett neues Array ist und keine Verbindung mehr zu den Arrays *\$array1* und *\$array2* hat, wirken sich nachträgliche Änderungen auf *\$array1* oder *\$array2* nicht mehr auf *\$array3* aus.

Wie das Beispiel zeigt, kann man auf diese Weise Befehlsergebnisse kombinieren. Obwohl *Get-ChildItem* (kurz *Dir*) nur nach einer einzelnen Dateierweiterung filtern kann, gelang es dem Beispiel oben, sowohl Logbuchdateien als auch ausführbare Dateien aus dem Windowsordner in einer Liste auszugeben.

Auf ähnliche Weise könnten Sie mit *Get-EventLog* Ereignislogbucheinträge aus unterschiedlichen Logbüchern zuerst separat abrufen und dann in einer gemeinsamen Liste zusammenfassen und ausgeben.

Arrays miteinander vergleichen

Sie möchten zwei Arrays miteinander vergleichen und herausfinden, welche Elemente unterschiedlich sind.

Lösung

Setzen Sie *Compare-Object* ein. Die folgende Zeile ermittelt die Elemente, die nur in *\$array1* enthalten sind:

```
PS > $array1 = 1,2,4,7,12,100,200
PS > $array2 = 2,3,7,12,13,14,100
PS > Compare-Object $array1 $array2 -passThru | Where-Object {
    { $_.SideIndicator -eq '<=' }
1
4
200
```

Möchten Sie nur die Elemente, die in *\$array2* enthalten sind, drehen Sie den »SideIndicator« um:

```
PS > Compare-Object $array1 $array2 -passThru | Where-Object {
    { $_.SideIndicator -eq '>=' }
3
13
14
```

Und interessieren Sie sich für die Elemente, die in beiden Arrays enthalten sind, prüfen Sie auf Gleichheit:

```
PS > Compare-Object $array1 $array2 -includeEqual -passThru | Where-Object {
    { $_.SideIndicator -eq '==' }
2
7
12
100
```

Hintergrund

Compare-Object vergleicht zwei Listen miteinander und ermittelt normalerweise nur die Unterschiede. Möchten Sie auch die Übereinstimmungen sehen, setzen Sie den Parameter *-include-*

Equal ein. Das Ergebnis von *Compare-Object* ist eine Liste, die in der Spalte *SideIndicator* anzeigt, ob das jeweilige Element in der ersten Liste, der zweiten Liste oder in beiden vorkommt.

Geben Sie den Parameter *-PassThru* an, ändert sich das Ergebnis. Jetzt werden die Originaldaten zurückgegeben, die allerdings nach wie vor über eine (jetzt aber normalerweise nicht mehr sichtbare) Eigenschaft *SideIndicator* verfügen. Mit *Where-Object* kann man so diejenigen Werte auswählen, mit denen man weiterarbeiten möchte.

Enthalten die beiden Listen keine einfachen Datentypen wie Texte oder Zahlen, sondern Objekte, dann müssen Sie *Compare-Object* mit dem Parameter *-property* die Spalte(n) nennen, die in den Vergleich einbezogen werden sollen. Möchten Sie zum Beispiel herausfinden, worin sich die Dienstekonfiguration zweier Computer unterscheidet, könnten Sie einen Schnappschuss vom lokalen System und einem zweiten System anlegen (hier: *storage1*) und dann vergleichen:

```
PS > $eigenedienste = Get-Service
PS > $vergleichsdienste = Get-Service -ComputerName storage1
PS > Compare-Object $eigenedienste $vergleichsdienste -property Name, →
    Status -PassThru | Sort-Object DisplayName | Select MachineName, →
    Status, DisplayName, Name | Format-Table -AutoSize
```

MachineName	Status	Name	DisplayName
.	Stopped	AeLookupSvc	Anwendungserfahrung
storage1	Running	AeLookupSvc	Anwendungskompatibilitäts-Suchdienst
storage1	Stopped	AlertSvc	Warndienst
storage1	Stopped	BITS	Intelligenter Hintergrundübertragungsdienst
.	Running	BITS	Intelligenter Hintergrundübertragungsdienst
storage1	Running	HidServ	HID Input Service
.	Stopped	hidserv	Zugriff auf Eingabegeräte
storage1	Running	MSDTC	Distributed Transaction Coordinator
.	Stopped	MSDTC	Distributed Transaction Coordinator

HINWEIS

Dieses Beispiel setzt voraus, dass Sie auf einen zweiten Computer remote zugreifen können, also über die notwendigen Berechtigungen und technischen Voraussetzungen verfügen. Remotezugriffe sind aber nicht unbedingt erforderlich. Sie könnten auch auf ein- und demselben Computer zwei »Schnappschüsse« generieren, um einen Vorher-Nachher-Zustand zu analysieren, beispielsweise vor und nach der Installation neuer Software. Dies ist auch über Neustarts hinweg möglich, wenn Sie den Schnappschuss jeweils mit *Export-CliXML* als XML-Datei speichern und später mit *Import-CliXML* wieder laden. Entscheidend ist die Auswahl der Eigenschaften, die *Compare-Object* vergleichen soll. Im Falle der Dienste waren dies die Eigenschaften *Name* und *Status*, damit Unterschiede der Dienststatus gefunden werden. Wollen Sie dagegen den Inhalt eines Ordners analysieren, lauten die Vergleichseigenschaften *Name* und *Length* oder sogar *Name*, *Length* und *LastWriteTime*.

Die Spalte *MachineName* gibt hierbei den Namen des jeweiligen Rechnersystems an, wobei ».« für den lokalen Computer steht. Kommt ein Dienstname nur einmal vor, ist dieser Dienst auf dem anderen Computer nicht vorhanden. Kommt der Dienst doppelt vor, unterscheidet sich sein Status. Er ist also auf dem einen Computer gestartet und auf dem anderen nicht.

Schlüssel-Wert-Paare speichern

Sie möchten Informationen mit einem Schlüssel versehen, damit Sie diese Informationen später schnell und einfach über ihren Schlüssel abrufen können.

Lösung

Setzen Sie Hashtables (assoziative Arrays) ein. Hashtables werden mit »@{ }« angelegt und bestehen aus Schlüssel-Wort-Paaren. Über den Schlüssel kann der Wert später wieder abgerufen werden.

```
PS > $person = @{name='Weltner'; vorname='Tobias'; alter=12}
PS > $person
```

Name	Value
----	-----
name	Weltner
vorname	Tobias
alter	12

```
PS > $person['Name']
Weltner
PS > $person.Name
Weltner
```

Hintergrund

Hashtables sind im Grunde herkömmliche eindimensionale Arrays. Allerdings verwenden Hashtables keinen numerischen Index für ihre Elemente, sondern Schlüsselwörter. Sie sprechen die Elemente in einer Hashtable deshalb nicht über Indexzahlen an, sondern über die zugeordneten Schlüssel.

Das Hashtable selbst wird über @{ } angelegt. Beachten Sie die geschweiften Klammern. Wären es runde Klammern, würden Sie ein normales Array anlegen. Innerhalb der geschweiften Klammern geben Sie die Schlüssel-Wert-Paare getrennt durch Semikola an.

Eine Besonderheit von Hashtables ist, dass Sie einen Wert sowohl über die von Arrays her bekannte Schreibweise mit den eckigen Klammern als auch über die von Objekten her bekannte Punkt-Schreibweise ansprechen können:

```

PS > $person = @{name='Weltner'; vorname='Tobias'; alter=12}
PS > $person['Name']
Weltner
PS > $person.Name
Weltner
PS > $info = 'Name'
PS > $person[$info]
Weltner
PS > $person.$info
Weltner

```

Dabei darf der Name des Schlüssels sogar in einer anderen Variablen enthalten sein. Geben Sie ein Hashtable aus, formatiert PowerShell diese automatisch spaltenweise in den Spalten *Name* und *Value*:

```

PS > $person

Name                               Value
----                               -
name                               Weltner
vorname                           Tobias
alter                             12

```

Über die Eigenschaft *Keys* erhält man sämtliche Schlüssel, die in einem Hashtable gespeichert sind, und könnte auf diese Weise auch programmgesteuert alle Werte abfragen, die in einem Hashtable gespeichert sind:

```

PS > Foreach ($schluessel in $person.Keys) { "Schlüssel '$schluessel' →
enthält den Wert '$($person.$schluessel)'" }
Schlüssel 'name' enthält den Wert 'Weltner'
Schlüssel 'vorname' enthält den Wert 'Tobias'
Schlüssel 'alter' enthält den Wert '12'

```

Hashtables können auch programmgesteuert angelegt werden. Dazu legen Sie zunächst ein leeres Hashtable an und verwenden dann die Methode *Add()*, um Schlüssel-Wert-Paare hinzuzufügen. Mit *ContainsKey()* finden Sie heraus, ob es einen bestimmten Schlüssel bereits im Hashtable gibt, denn doppelte Schlüssel sind nicht erlaubt:

```

PS > $array = @{}
PS > $array.Add('Name', 'Weltner')
PS > $array.ContainsKey('Name')
True

```

Hashtables werden in den folgenden Szenarien eingesetzt:

- **Universeller Datencontainer** Ermöglicht Ihnen, Informationen als Schlüssel-Wert-Paare übersichtlich abzulegen

- **Objektgenerierung** In Verbindung mit *New-Object* und dem Parameter *-Property* kann ein Hashtable als Prototyp für ein neues Objekt verwendet werden. Die Schlüssel im Hashtable werden dann zu Objekteigenschaften und die Werte zu ihren Inhalten.
- **Spaltenformatierung** Viele Cmdlets, die über den Parameter *-Property* verfügen, akzeptieren anstelle eines Spaltennamens (Eigenschaftennamens) auch ein Hashtable, das dann die Schlüssel *Name* und *Expression* enthalten muss. *Name* legt den Spaltennamen fest und *Expression* muss ein Skriptblock liefern, der den Inhalt der Spalte dynamisch berechnet:

```
PS > $spalte = @{Name='Größe'; Expression='{0:0.0} MB' -f ($_.Length/1MB) }}
PS > Dir $env:windir | Format-Table Name, $spalte
```

- **Splatting** Die Parameter eines Cmdlets dürfen mithilfe eines Hashtables übergeben werden. Dabei entspricht jeder Schlüssel im Hashtable dem gewünschten Parameternamen und jeder Wert wird dem entsprechenden Parameter übergeben.

```
PS > $MailMessage = @{
>> To = "someone@somecompany.com"
>> From = "me@mycompany.com"
>> Subject = "Hello There"
>> Body = "This mail was sent by PowerShell using Splatting mechanisms"
>> Smtpserver = "smtphost"
>> ErrorAction = "SilentlyContinue"
>> }
>>

PS > Send-MailMessage @MailMessage
```

Im nächsten Beispiel werden Parameter als Hashtable an den Befehl *Dir* übergeben, der daraufhin den Windows-Ordner rekursiv nach Dateien mit der Erweiterung *.log* durchsucht und Fehlermeldungen unterdrückt. Hier wird das Hashtable im Gegensatz zum letzten Beispiel in einer einzelnen Zeile angelegt:

```
PS > $p = @{Path="$env:windir"; filter='*.log'; Erroraction='SilentlyContinue'; →
Recurse=$true}
PS > Dir @p
```

HINWEIS

Damit ein Cmdlet ein Hashtable als Parametereingabe akzeptiert, muss der Name des Hashtables mit einem vorangestellten »@« und nicht mit dem üblichen Variablenzeichen »\$« angegeben werden.

Schlüssel-Wert-Paare sortieren

Sie haben eine Liste mit Schlüssel-Wert-Paaren angelegt, die Sie nun sortieren möchten.

Lösung

Eine Liste mit Schlüssel-Wert-Paaren enthält zwei Eigenschaften, nämlich *Name* (die Schlüssel) und *Value* (die den Schlüsseln zugeordneten Werte). Setzen Sie *Sort-Object* ein und sortieren Sie entweder nach *Name* oder nach *Value*.

```
PS > $array = @{name='Weltner'; vorname='Tobias'; alter=8}
PS > $array.GetEnumerator() | Sort-Object name
Name      Value
----      -
alter     8
name      Weltner
vorname   Tobias

PS > $array.GetEnumerator() | Sort-Object value
Name      Value
----      -
alter     8
vorname   Tobias
name      Weltner
```

Hintergrund

Um ein Hashtable sortieren zu können, kann man nicht einfach dessen Inhalt an *Sort-Object* übergeben. Würde man das tun, würde *Sort-Object* lediglich das Hashtableobjekt empfangen, aber nicht die in dem Hashtable enthaltenen Schlüssel-Wert-Paare, und könnte deshalb den Inhalt des Hashtables nicht sortieren. Das folgende Beispiel zeigt, was *Sort-Object* tatsächlich empfangen würde:

```
PS > $array = @{name='Weltner'; vorname='Tobias'; alter=8}
PS > $array | ForEach-Object { $_.GetType().name }
Hashtable
```

Jedes Hashtable enthält jedoch die Methode *GetEnumerator()*. Diese Methode liefert die einzelnen im Hashtable gespeicherten Schlüssel-Wert-Paare zurück, die jetzt von *Sort-Object* wie jedes andere Objekt auch sortiert werden können.

```
PS > $array.GetEnumerator() | ForEach-Object { $_.GetType().name }
DictionaryEntry
DictionaryEntry
DictionaryEntry
```

Zusammenfassung

In diesem Kapitel haben Sie fünf verschiedene Arraytypen kennengelernt (Tabelle 3.1). Das einfache eindimensionale Array spielt dabei die Hauptrolle. Es wird von PowerShell immer dann eingesetzt, wenn ein Befehl mehr als ein Ergebnis liefert oder wenn Sie mit kommaseparierten Listen eigene Arrays erstellen.

Das Hashtable nutzt PowerShell immer dann, wenn die Elemente im Array über einen »sprechenden« Schlüsselnamen erreichbar sein soll.

Die übrigen Arraytypen werden von PowerShell selbst nicht genutzt. Sie können diese Arraytypen aber in eigenen Skripts einsetzen, um von deren individuellen Vorzügen zu profitieren.

Arraytyp	Beschreibung
<i>Einfaches eindimensionales Array</i>	Speichert mehrere Elemente mit einem numerischen eindimensionalen Index. Die einzelnen Objektelemente sind nicht typisiert, dürfen also beliebige Daten speichern. Dies ist das Standardarray von PowerShell und entspricht dem .NET-Typ <code>[Object[]]</code> .
<i>Typisierte Arrays</i>	Wie vor, jedoch mit einem fest vorgegebenen Datentyp. Das Array speichert also nur noch eine »Sorte« von Informationen, zum Beispiel Ganzzahlen.
<i>Klassische mehrdimensionale Arrays</i>	Diese symmetrischen Arrays besitzen zwei oder mehr Dimensionen und werden von PowerShell normalerweise nicht eingesetzt. Man kann solche Arrays aber über New-Object direkt von .NET Framework anfordern.
<i>ArrayList</i>	Besonderer Arraytyp, in den jedes normale Array verwandelt werden kann. Eine ArrayList stellt Methoden zur Verfügung, um schnell und effizient bestehende Arrayelemente zu streichen oder neue Elemente an beliebiger Position ins Array einzufügen.
<i>Hashtable</i>	Array verwendet anstelle eines numerischen Index beliebige Schlüsselwörter für die Arrayelemente. Hashtables werden über die Konstruktion <code>@{}</code> angelegt und spielen in PowerShell eine große Rolle, da sie auch intern für verschiedene Techniken wie dynamische Spaltenberechnung und Splatting eingesetzt werden.

Tabelle 3.1 Arraytypen in diesem Kapitel

Außerdem haben Sie in diesem Kapitel zwei .NET-Typen kennengelernt. Normale Standardarrays entsprechen dem .NET-Typ `[Object[]]`. Der Typ `[System.Collection.ArrayList]` stellt ein Array mit zusätzlichen Funktionen bereit, über die Sie sehr einfach und effizient nachträglich Elemente aus dem Array entfernen oder in das Array einfügen können.

Kapitel 4

PowerShell-Pipeline

In diesem Kapitel:

Informationen filtern	123
Objekteigenschaften (Spalten) auswählen	127
Ergebnisse einzeln verarbeiten	129
Ausgaben sortieren	133
Ergebnisse gruppieren	136
Ergebnisse in Text umwandeln	141
Zusammenfassung	143

Die PowerShell-Pipeline ist eines der wichtigsten Konzepte der PowerShell, mit dem sich mehrere Befehle zu einer Befehlskette zusammenfassen lassen. Obwohl der Aufbau und die Länge einer solchen »Pipeline« im Einzelfall ganz unterschiedlich sein kann, folgen Pipelines jedoch stets demselben Grundmuster:

```
Get-* | *-Object | [Format-*] | Out-*
Get-* | *-Object | Export-*
```

Die Pipeline funktioniert also im Grunde wie ein Fabrikfließband, auf dem nach und nach aus den Rohdaten die gewünschten Ergebnisse geformt werden. Sie beginnt deshalb in aller Regel mit einem Befehl, der Daten liefert, also entweder einem Cmdlet, dessen Name mit »Get« beginnt, oder mit den Rohdaten selbst, die beispielsweise aus einer Variable stammen können.

Danach werden so viele »Fabrikroboter« wie nötig mit dem Pipeline-Symbol »|« angefügt. Sie erhalten jeweils das Ergebnis vom vorangegangenen Befehl, erledigen ihre Arbeit und geben das eigene Ergebnis dann an den folgenden Befehl weiter. Diese Cmdlets stammen aus der Gruppe der **-Object*-Cmdlets:

Name	Beschreibung
<i>Select-Object</i>	Wählt die Eigenschaften (Spalten) aus, die sichtbar sein sollen
<i>Sort-Object</i>	Sortiert das Ergebnis basierend auf einer oder mehreren Eigenschaften
<i>Group-Object</i>	Gruppiert die Ergebnisse basierend auf einer oder auf mehreren Eigenschaften
<i>Measure-Object</i>	Zählt die Ergebnisse und kann optional numerische Eigenschaften auswerten
<i>ForEach-Object</i>	Führt einen Skriptblock für jedes einzelne Ergebnis aus. Innerhalb des Skriptblocks repräsentiert <code>\$_</code> das gerade über die Pipeline laufende Objekt. Dieses Cmdlet entspricht also einer Schleife.
<i>Where-Object</i>	Führt einen Skriptblock für jedes einzelne Ergebnis aus. Innerhalb des Skriptblocks repräsentiert <code>\$_</code> das gerade über die Pipeline laufende Objekt. Ergibt der Skriptblock den Wert <code>\$true</code> , wird das Objekt zum nächsten Pipelinebefehl weitergeleitet, sonst nicht. Dieses Cmdlet entspricht also einer Bedingung.

Tabelle 4.1 Wichtige Cmdlets innerhalb der PowerShell-Pipeline

Am Ende der Pipeline können die Ergebnisse mithilfe von Cmdlets der Gruppe *Out-** entweder in Text verwandelt und ausgegeben, mit Cmdlets der Gruppe *Export-** als Objekte serialisiert oder mit Cmdlets der Gruppe *ConvertTo-** in andere Formate konvertiert werden. Setzen Sie keine dieser Cmdlets ein, werden die Ergebnisse automatisch mit *Out-Default* in Text konvertiert und in die Konsole ausgegeben.

Wichtig hervorzuheben ist, dass die Pipeline die Daten in der Regel in Echtzeit bearbeitet. Sobald der erste Befehl in der Pipeline ein Ergebnis ausgibt, wird dieses durch die Pipeline geschickt und von den einzelnen Befehlen weiterbearbeitet. Noch während der erste Befehl also weitere Ergebnisse liefert, sind die übrigen Befehle in der Pipeline bereits damit beschäftigt, die schon gelieferten Ergebnisse zu bearbeiten. Zu Blockierungen kommt es nur, wenn Sie Befehle

einsetzen, die zuerst alle Ergebnisse sammeln, beispielsweise, weil die Ergebnisse anschließend sortiert werden sollen.

Informationen filtern

Sie möchten die Ergebnisse eines Befehls nach bestimmten Kriterien filtern.

Lösung

Leiten Sie die Ergebnisse an *Where-Object* weiter und legen Sie eine Bedingung fest. Die Bedingung wird in Form eines Filterskriptblocks formuliert. Darin repräsentiert die Variable `$_` das jeweils zu untersuchende Objekt.

Um alle Dateien im Windows-Ordner zu finden, die größer sind als 1 Mbyte, werten Sie beispielsweise die Eigenschaft *Length* aus:

```
PS > dir $env:windir | Where-Object { $_.Length -gt 1MB }
```

Möchten Sie alle Prozesse ermitteln, die mehr als 20 Sekunden Prozessorzeit verbraucht haben, lassen Sie die Eigenschaft *CPU* auswerten:

```
PS > Get-Process | Where-Object { $_.CPU -gt 20 }
```

Die folgende Zeile liefert nur Prozesse der Firma »Microsoft«:

```
PS > Get-Process | Where-Object { $_.Company -like '*microsoft*' }
```

Wollen Sie alle Dienste finden, die zurzeit nicht ausgeführt werden, überprüfen Sie die Eigenschaft *Status*:

```
PS > Get-Service | Where-Object { $_.Status -eq 'Stopped' }
```

Und möchten Sie aus einem Ordner nur die Dateien sehen, aber nicht die Unterordner, greifen Sie auf die Eigenschaft *PSIsContainer* zu:

```
PS > dir | Where-Object { $_.PSIsContainer -eq $false }
```

Ausschließlich Unterordner, aber keine Dateien, erhalten Sie so:

```
PS > dir | Where-Object { $_.PSIsContainer -ne $false }
```

Möchten Sie aus einem textbasierten Logbuch nur diejenigen Zeilen herausfiltern, die ein bestimmtes Stichwort enthalten, verfahren Sie so:

```
PS > Get-Content $env:windir\windowsupdate.log -Encoding UTF8 | Where-Object { $_ -like '*successfully installed*' }
```

Und möchten Sie alle Updates sehen, deren KB-Artikelnummer mit »KB98« beginnt, werten Sie die Eigenschaft *HotfixID* aus:

```
PS > Get-Hotfix | Where-Object { $_.HotfixID -like 'KB98*' }
```

Auch das Ergebnis nativer Befehle lässt sich so filtern. Der folgende Code verwendet den nativen Befehl *ipconfig.exe* und gibt nur solche Zeilen zurück, in denen das Stichwort »IP« vorkommt:

```
PS > ipconfig | Where-Object { $_ -like '*IP*' }
```

Hintergrund

Where-Object untersucht jedes einzelne Objekt, das durch die Pipeline gesendet wird. Der Skriptblock, den Sie angeben, wird also für jedes Ergebnisobjekt erneut ausgewertet. Das gerade untersuchte Ergebnisobjekt wird darin durch die Variable *\$_* repräsentiert. Ergibt der Skriptblock das Ergebnis *\$true*, gilt das Objekt als erwünscht und wird nicht herausgefiltert, andernfalls schon.

Liefert *Get-Process* beispielsweise 20 laufende Prozesse und leiten Sie das Ergebnis an *Where-Object* weiter, wird der Skriptblock hinter *Where-Object* zwanzig Mal ausgeführt. Da *Get-Process* als Ergebnis *Process*-Objekte liefert, die über zahlreiche unabhängige Eigenschaften verfügen, wählen Sie im Skriptblock hinter dem Ausdruck »*\$_*« die Eigenschaft aus, die Sie überprüfen möchten. Welche Eigenschaften dabei zur Verfügung stehen, kann *Get-Member* für Sie ermitteln:

```
PS > Get-Process | Get-Member -memberType *property
```

Handelt es sich dagegen beim Objekt um einen primitiven Datentyp (einfache Zahlen oder Texte), kann *\$_* direkt für den Vergleich herangezogen werden. Sie brauchen dann also nicht mit der Punktschreibweise hinter *\$_* eine bestimmte Objekteigenschaft anzugeben. Die nächste Zeile würde beispielsweise nur Zahlen auflisten, die kleiner sind als 6:

```
PS> 1..10 | Where-Object { $_ -lt 6 }  
1  
2  
3  
4  
5
```

Möchten Sie mehrere Kriterien kombinieren, verwenden Sie mehrere hintereinandergeschaltete *Where-Object*-Anweisungen. Dies entspricht einer logischen »Und«-Verknüpfung. Die folgende Zeile findet beispielsweise alle Prozesse der Firma Microsoft, die mehr als 20 Sekunden CPU-Zeit beansprucht haben:

```
PS > Get-Process | Where-Object { $_.CPU -gt 20 } | Where-Object {  
    { $_.Company -like '*Microsoft*' }  
}
```

Alternativ könnten Sie auch den logischen Operator *–and* einsetzen und mit seiner Hilfe mehrere Bedingungen kombinieren. Das Hintereinanderschalten mehrerer *Where-Object*-Klauseln ist aber in den meisten Fällen empfehlenswerter, weil dadurch einfacherer Code entsteht und Sie sehr einfach nachträglich eine Bedingung wieder entfernen oder eine weitere hinzufügen können.

Wollen Sie dafür sorgen, dass alle Ergebnisse angezeigt werden, bei denen entweder die eine oder die andere Bedingung erfüllt ist, sind Sie allerdings gezwungen, mehrere Bedingungen mit dem logischen Operator *–or* zu kombinieren. Die folgende Zeile liefert alle Textzeilen einer Logbuchdatei, die entweder das Stichwort »No Network Connectivity« oder »Update is not allowed« enthalten:

```
PS > Get-Content $env:windir\windowsupdate.log | Where-Object {  
    { ($ -like '*no network connectivity*') -or ($ -like '*update is not allowed*') }  
}
```

Diese Zeile liefert selbstverständlich keine Resultate, wenn die Logbuchdatei die gesuchten Stichwörter nicht enthält (oder Sie sich vertippt haben). Nutzen Sie den Operator *–cli* anstelle von *–like*, wenn Sie zwischen Groß- und Kleinschreibung unterscheiden müssen.

Eine Besonderheit ist das Filtern leerer Eigenschaften. Möchten Sie beispielsweise nur die Prozesse sehen, bei denen die Eigenschaft *Company* gefüllt ist, formulieren Sie so:

```
PS > Get-Process | Where-Object { $_.Company -eq $null }  
...  
PS > Get-Process | Where-Object { $_.Company -eq $null } | Select-Object Name, Company  
...  
...
```

Vollkommen leere Eigenschaften werden durch die vordefinierte Variable *\$null* repräsentiert. Hin und wieder werden Sie vielleicht auch Codezeilen wie die folgende sehen:

```
PS > Get-Process | Where-Object { $_.Company } | Select-Object Name, Company
```

Hierbei macht man sich zunutze, dass ein vollkommen leerer Wert (also beispielsweise eine nicht festgelegte Objekteigenschaft) bei der Umwandlung in einen booleschen Wert zu *\$false* wird, während jeder beliebige sonstige Datenwert (mit Ausnahme des Zahlenwerts 0) zu *\$true* konvertiert wird. Um also Objekte auszufiltern, die in einer bestimmten Eigenschaft keinen definierten Wert besitzen, geben Sie im Skriptblock hinter »\$_.« den Namen dieser Eigenschaft an.

So erhalten Sie beispielsweise durch den folgenden Filter nur diejenigen Netzwerkadapter, bei denen tatsächlich eine IP-Adresse zugewiesen ist:

```
PS > Get-WmiObject Win32_NetworkAdapterConfiguration | Where-Object {  
    { $_.IPAddress }  
}
```

Allerdings funktioniert diese Abkürzung nur, wenn die Eigenschaft, die Sie im Skriptblock auswerten, niemals den Wert 0 annehmen kann. Dieser Wert entspricht nämlich ebenfalls dem booleschen Wert *\$false*, sodass Sie nicht nur vollkommen leere Eigenschaften ausfiltern, sondern auch solche, die genau den Wert 0 beinhalten.

HINWEIS

Die Filterung mit *Where-Object* erfolgt clientseitig, was besonders bei Remotezugriffen große Bedeutung hat. Da bei einer clientseitigen Filterung zuerst alle Daten zum Client transportiert werden müssen, ist diese Filterung nicht sehr effizient und sollte nur dann eingesetzt werden, wenn eine serverseitige Filterung nicht möglich ist. Eine serverseitige Filterung setzt voraus, dass das Cmdlet, das die Daten abrufen, selbst mithilfe eines Parameters die Datenmenge reduzieren kann. Im Falle der WMI verfügt *Get-WmiObject* dazu über den Parameter *-Filter*, sodass Sie die Netzwerkkarten auch über folgende Zeile effizienter serverseitig hätten filtern können:

```
PS > Get-WmiObject Win32_NetworkAdapterConfiguration -filter 'IPEnabled=true'
```

Dabei wird nicht etwa geprüft, ob die Eigenschaft *IPAddress* null ist, sondern auf eine andere, ebenso aussagekräftige Eigenschaft namens *IPEnabled* zurückgegriffen. Sie ist *\$true*, wenn der jeweiligen Netzwerkkonfiguration eine IP-Adresse zugewiesen ist, sonst *\$false*. Da es sich bei der serverseitigen Filterung um eine Leistung des Cmdlets und nicht der PowerShell handelt, übergeben Sie dabei keinen PowerShell-Code, sondern müssen sich nach dem Datentyp richten, den der zuständige Parameter von Ihnen erwartet. Der Parameter *-Filter* des Cmdlets *Get-WmiObject* erwartet beispielsweise eine Filterung im WQL-Format, also der Abfragesprache der WMI.

Aber auch ohne Remotezugriffe ist die serverseitige Filterung mittels Cmdlet-Parameter stets schneller. Die folgende Zeile liefert alle Ereignislogbucheinträge vom Typ *Error* aus dem Logbuch *System* und verwendet dazu die langsame clientseitige Filterung:

```
PS > Get-EventLog -LogName System | Where-Object { $_.EntryType -eq 'Error' }
```

Mehr als doppelt so schnell (und sehr viel simpler) ist die serverseitige Filterung, die sich zunutze macht, dass das Cmdlet *Get-EventLog* einen Parameter namens *-EntryType* besitzt:

```
PS > Get-EventLog -LogName System -EntryType Error
```

Grundsätzlich sollten Sie also zuerst versuchen, die Ergebnisse mithilfe der Parameter zu filtern, die das Cmdlet anbietet, das die Daten für Sie beschafft. Viele Cmdlets unterstützen Parameter, die genauso heißen wie die Spalten der Ergebnisdaten und mit denen man die Ergebnisse nach dieser Spalte filtern kann. Hier weitere Beispiele:

```
PS > Get-Command *service* -CommandType Cmdlet
PS > Get-Alias -Definition Get-ChildItem
PS > Get-Process | Get-Member -memberType *Property
PS > Get-ExecutionPolicy -Scope CurrentUser
```

Objekteigenschaften (Spalten) auswählen

Sie möchten nur bestimmte Eigenschaften (Spalten) eines Objekts weiterbearbeiten und die übrigen entfernen.

Lösung

Leiten Sie das Ergebnis des Befehls an *Select-Object* weiter und geben Sie mit dem Parameter *-property* an, welche Spalten Sie sehen möchten:

```
PS > Get-Process | Select-Object Name, *memory*

Name                : AcroRd32
NonpagedSystemMemorySize : 17960
NonpagedSystemMemorySize64 : 17960
PagedMemorySize      : 68947968
PagedMemorySize64    : 68947968
PagedSystemMemorySize : 342640
PagedSystemMemorySize64 : 342640
...

PS > Get-Process | Select-Object Name, CPU, PeakVirtualMemorySize

Name                CPU                PeakVirtualMemorySize
----                -
AcroRd32            14,6328938      230084608
alg                 0                29872128
AppleMobileDeviceService 0,6552042      96903168
ApplicationUpdater  0,156001        71254016
...
```

PowerShell bestimmt dabei automatisch die Art der Formatierung: Geben Sie vier oder weniger Eigenschaften an, wird das Ergebnis als Tabelle nebeneinander angezeigt, andernfalls als Liste untereinander.

Hintergrund

PowerShell enthält mit dem Extended Type System (ETS) einen Mechanismus, der normalerweise vollautomatisch die wichtigsten Eigenschaften eines Objekttyps für Sie auswählt. Diese werden als Tabelle nebeneinander angezeigt, wenn es sich um vier oder weniger handelt, andernfalls als Liste untereinander. Von dieser Regel kann in Ausnahmefällen abgewichen werden, wenn im ETS abweichende Regeln für den Objekttyp hinterlegt worden sind.

Möchten Sie dagegen selbst bestimmen, welche Eigenschaften angezeigt werden, setzen Sie *Select-Object* ein und geben die Namen der gewünschten Eigenschaften als kommaseparierte Liste an. Ob die Ergebnisse als Tabelle nebeneinander oder als Liste untereinander angezeigt werden, hängt auch hier von der Anzahl der ausgewählten Eigenschaften ab.

Möchten Sie sämtliche Eigenschaften eines Objekts sehen, geben Sie als Eigenschaftennamen den Platzhalter *»*«* an:

```
PS > Get-Process powershell | Select-Object *
```

WICHTIG

Meistens entspricht die Spaltenüberschrift dem Namen der zugrundeliegenden Objekteigenschaft – aber nicht immer. Führen Sie *Get-Process* aus, wird eine Spalte namens »CPU(s)« genannt:

```
PS > Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
244	23	58092	71408	208	17,18	548	AcroRd32
76	8	1412	208	27		2656	alg
...							

Dies entspricht aber nicht dem wahren Spaltennamen, und wenn Sie *Select-Object* beauftragen, die Spalte »CPU(s)« anzuzeigen, kommt es sogar zu einem Fehler. Hier hat das ETS aus Gründen der besseren Lesbarkeit für die Spaltenüberschrift einen anderen, abweichend Namen festgelegt. Die wahren Eigenschaftennamen erfahren Sie immer mit *Select-Object **. Wie sich herausstellt, heißt die Eigenschaft in Wirklichkeit *CPU* und nicht *CPU(s)*.

```
PS > Get-Process | Select-Object *
```

```
...
Company           : Adobe Systems Incorporated
CPU                : 17,1757101
FileVersion        : 9.4.0.195
ProductVersion     : 9.4.0.195
...
```

Ähnlich ist es bei *Get-EventLog*:

```
PS > Get-EventLog System -Newest 2
```

Index	Time	EntryType	Source
175275	Jan 31 21:03	Information	Service Control M...
175274	Jan 31 21:03	Information	Service Control M...

Die Spalte *Time* heißt bei näherer Untersuchung in Wirklichkeit *TimeGenerated*. Bei den mitgelieferten Standard-Cmdlets sind solche Abweichungen nur selten. Bei einigen Befehlserweiterungen wie zum Beispiel den Cmdlets für Microsoft Exchange finden sich abweichenden Spaltenüberschriften dagegen häufig.

Alternativ können Sie auch *Get-Member* beauftragen, die Eigenschaften eines Objekttyps sichtbar zu machen:

```
PS > Get-Process powershell | Get-Member -memberType *Property
```

Platzhalterzeichen dürfen auch innerhalb von Eigenschaftennamen eingesetzt werden. Die folgende Zeile liefert die aktuelle Bildschirmauflösung eines physischen Systems (WMI liefert diese Informationen nicht für virtuelle Maschinen). Anstelle die beiden gewünschten Eigenschaften

vollständig als kommaseparierte Liste anzugeben (*CurrentHorizontalResolution*, *CurrentVerticalResolution*), wird nur die gemeinsame Endung **Resolution* festgelegt.

```
PS > Get-WmiObject Win32_VideoController | Select-Object *Resolution
```

CurrentHorizontalResolution	CurrentVerticalResolution
----- 1680	----- 1050

HINWEIS

Möchten Sie selbst bestimmen, ob die Informationen als Tabelle oder als Liste dargestellt werden, greifen Sie anstelle von *Select-Object* zu *Format-Table* oder *Format-List*. Weil diese Cmdlets allerdings die Informationen umwandeln in Formatierungsobjekte, die nur noch von der Konsole korrekt dargestellt werden können, müssen *Format-**-Cmdlets stets am Ende einer Pipeline eingesetzt werden und dürfen höchstens noch an *Out-**-Cmdlets weitergeleitet werden. *Format-Table* hat eine besondere Bedeutung, wenn Spalteninhalte aus Platzgründen nicht vollständig in der Konsole angezeigt werden können. In diesem Fall kürzt PowerShell einzelne Spalteninhalte mit »...« ab. Um die Spalteninhalte möglichst vollständig anzuzeigen, verwendet man den Parameter *-AutoSize*:

```
PS > Get-Hotfix | Format-Table -AutoSize
```

Bei Angabe von *-AutoSize* sammelt *Format-Table* alle Ergebnisse zuerst, um danach die optimale Spaltenbreite zu bestimmen. So geht der Echtzeitcharakter der Pipeline zugunsten einer besseren Darstellung verloren. Auch mit *-AutoFormat* kann der Platz in der Konsole jedoch zu schmal sein, um Tabellen ungekürzt anzuzeigen. In diesem Fall kann der Parameter *-Wrap* helfen: Dieser bricht zu breite Spalteninhalte in mehrere Zeilen um:

```
PS > Get-Hotfix | Format-Table -AutoSize -Wrap
```

Möchten Sie Ergebnisse ungekürzt in eine Textdatei schreiben, verwenden Sie eine Kombination aus *Format-Table* und *Out-File*. Geben Sie bei *Out-File* den Parameter *-Width* an, damit PowerShell mehr Platz für die Ausgabe nutzen kann. Die folgende Zeile schreibt die Ergebnisse in eine Textdatei, die maximal 10.000 Zeichen breit sein darf, aber tatsächlich nur so breit sein wird wie nötig, um keine Spalte zu kürzen.

```
PS > Get-EventLog System -EntryType Error -Newest 10 | Format-Table -AutoSize | →
    Out-File $home\report.txt -Width 10000
PS > Invoke-Item $home\report.txt
```

Ergebnisse einzeln verarbeiten

Sie möchten die Ergebnisse einer Pipeline einzeln in Echtzeit verarbeiten. Beispielsweise wollen Sie eine Liste mit Computernamen oder IP-Adressen anpingen.

Lösung

Setzen Sie *ForEach-Object* (Kurzform: »%«) ein. Das jeweilige Pipeline-Objekt steht in *\$_* zur Verfügung. Die folgende Zeile listet jede einzelne Datei eines Ordnerlistings auf und gibt die Eigenschaften *Name* und *LastWriteTime* aus:

```
PS > dir | ForEach-Object { '{0} wurde am {1} zuletzt geändert' -f $_.Name, $_.LastWriteTime }
Application Data wurde am 19.11.2007 21:57:06 zuletzt geändert
Auswertung wurde am 13.05.2008 12:43:47 zuletzt geändert
bin wurde am 30.05.2008 07:16:48 zuletzt geändert
Bluetooth Software wurde am 16.11.2007 23:46:24 zuletzt geändert
Contacts wurde am 16.11.2007 18:07:55 zuletzt geändert
Desktop wurde am 30.05.2008 14:10:41 zuletzt geändert
(...)
```

Die Pipeline-Daten können auch aus einer Datei gelesen werden. Legen Sie eine Datei namens *Serverliste.txt* mit Servernamen an. Die folgende Zeile würde die in der Datei aufgeführten Server der Reihe nach abfragen:

```
PS > Get-Content c:\daten\Serverliste.txt | ForEach-Object { Get-WmiObject Win32_OperatingSystem -computer $_ }
```

Ebenso leicht könnten Sie die in der Datei aufgeführten Server herunterfahren:

```
PS > Get-Content c:\daten\Serverliste.txt | ForEach-Object { Stop-Computer -Computername $_-whatif }
```

Woher die Daten stammen, die die Pipeline verarbeitet, spielt keine Rolle. Sie müssen nicht von einem Befehl stammen. Die folgende Zeile generiert aus einem Zahlenbereich eine Liste mit IP-Adressen:

```
PS > 1..255 | ForEach-Object { "192.168.1.$_" }
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
```

Diese IP-Adressen könnten an *Test-Connection* weitergeleitet werden, um zu überprüfen, welche Computer gerade online und erreichbar sind:

```
PS > 1..255 | ForEach-Object { "192.168.1.$_" } | ForEach-Object { Test-Connection -computername $_ -Count 1 }
```

HINWEIS

Falls diese Zeile Fehlermeldungen liefert, kann dies daran liegen, dass Sie augenblicklich nicht mit einem Netzwerk verbunden sind.

Auch formatierte Listen sind mithilfe des Formatierungsoperators »-f« möglich. Die folgende Zeile generiert eine Liste mit PC-Namen, wobei die laufende Nummer immer dreistellig ist:

```
PS > 1..100 | ForEach-Object { 'PC{0:000}' -f $_ }  
PC001  
PC002  
PC003  
...  
PC010  
PC011
```

Hintergrund

ForEach-Object erwartet einen Skriptblock, der für jedes Objekt ausgeführt wird, das vom vorangegangenen Befehl geliefert wurde. Innerhalb des Skriptblocks repräsentiert die Variable `$_` das jeweilige Objekt. Es liegt in Ihrer Verantwortung, innerhalb des Skriptblocks einen Rückgabewert zu liefern, den dann das nächstfolgende Cmdlet der Pipeline erhält. Im einfachsten Fall wird das empfangene Objekt in `$_` unverändert weitergereicht. So kann man beispielsweise Fortschrittsanzeigen realisieren:

```
PS > Dir $env:windir -filter *.log -recurse -ErrorAction SilentlyContinue | →  
ForEach-Object { Write-Progress 'Suche Logbuchdateien...' $_.FullName; $_ }
```

Eine Aktualisierung der Fortschrittsanzeige in Echtzeit erreichen Sie über die folgende Zeile, die dafür sehr viel langsamer arbeitet:

```
PS > Dir $env:windir -recurse -ErrorAction SilentlyContinue | ForEach-Object →  
{ Write-Progress 'Suche Logbuchdateien...' (Split-Path $_.FullName); $_ } →  
| Where-Object { $_.Extension -eq '*.log' }
```

Der Skriptblock muss allerdings nicht das empfangene Objekt unverändert weitergeben. Er kann es auch bearbeiten. Die folgende Zeile empfängt Textinformationen (IP-Adressen) und gibt DNS-Informationsobjekte zurück:

```
PS > '127.0.0.1', '127.0.0.2', '10.10.10.11' | ForEach-Object { $ip = $_; try →  
{ [System.Net.DNS]::GetHostByAddress($_) } catch { Write-Warning →  
"Konnte $ip nicht auflösen: $_" } }
```

Grundsätzlich arbeitet *ForEach-Object* als Schleife. Sie könnten dieses Cmdlet also auch dazu verwenden, um Codeblöcke zu wiederholen. Die folgende Zeile startet 10 Instanzen von Notepad:

```
PS > 1..10 | ForEach-Object { notepad.exe }
```

ForEach-Object kann auch dazu eingesetzt werden, um nicht-Pipeline-fähige Befehle in der Pipeline auszuführen. Der Parameter *-computername* des Cmdlets *Stop-Computer* kann beispielsweise Informationen über die Pipeline nur von Objekten empfangen, die eine *ComputerName*-Eigenschaft besitzen. Möchten Sie einen Rechnerpark aber lieber über eine Liste mit Rechnernamen herunterfahren, setzen Sie *ForEach-Object* ein. Die folgende Zeile würde alle Rechner, die in der angegebenen Textdatei untereinander angegeben sind, herunterfahren:

```
PS > Get-Content c:\rechnerliste.txt | ForEach-Object →
    { Stop-Computer -computername $_ -whatif }
```

Entfernen Sie den Parameter *-whatif*, wenn Sie die Rechner tatsächlich herunterfahren wollen. Ist ein Cmdlet dagegen in der Lage, die Informationen direkt über die Pipeline zu empfangen, setzen Sie *ForEach-Object* nicht ein, sondern leiten die Informationen direkt an das Cmdlet weiter. Die folgende Zeile funktioniert zwar, ist aber nicht effizient:

```
PS > Get-Process notepad | ForEach-Object { Stop-Process -name $_.Name }
```

Formulieren Sie stattdessen:

```
PS > Get-Process notepad | Stop-Process
```

Überprüfen Sie stets, ob eine Pipeline für die Aufgabe überhaupt erforderlich ist:

```
PS > Stop-Process -name notepad
```

ForEach-Object kann am Beginn und am Ende der Pipeline weitere Aufgaben ausführen, wenn Sie anstelle eines Skriptblocks drei Skriptblöcke angeben. Die nächste Zeile färbt alle *.exe*-Dateien eines Ordnerlistings rot. Der *begin*- und der *end*-Block werden dazu verwendet, die ursprünglichen Konsolenfarben zu speichern und wiederherzustellen:

```
PS > dir $env:windir | ForEach-Object -begin {
>> $farbeAlt = $host.UI.RawUI.ForegroundColor
>> } -process {
>> If ($_.Extension -eq '.exe') { $farbeNeu = 'Red' } else { $farbeNeu = 'Green' }
>> $host.UI.RawUI.ForegroundColor = $farbeNeu
>> $_
>> } -end {
>> $host.UI.RawUI.ForegroundColor = $farbeAlt
>> }
>>
```

ForEach-Object entspricht damit im Grunde einer Funktion mit den drei Skriptblöcken *begin*, *process* und *end*. Deshalb können Sie die Färbefunktion aus dem letzten Beispiel auch als Funktion formulieren:

```
PS > Function Markiere-Datei($extension='.exe') {
>> begin {
>> $farbeAlt = $host.UI.RawUI.ForegroundColor
>> }
>> process {
>> If ($_.Extension -eq $extension) { $farbeNeu = 'Red' } else →
>> { $farbeNeu = 'Green' }
>> $host.UI.RawUI.ForegroundColor = $farbeNeu
>> $_
>> }
>> end {
>> $host.UI.RawUI.ForegroundColor = $farbeAlt
>> }
>> }
>> }
```

Auf diese Weise heben Sie beliebige Dateien farblich hervor. Die nächste Zeile würde alle *.dll*-Dateien aus dem *System32*-Ordner farblich markieren:

```
PS > dir $env:windir\system32 | Markiere-Datei '.dll'
```

Der *begin*-Block wird ausgeführt, wenn die Pipeline startet. Hier merkt sich die Funktion die Ausgangsfarbe in *\$farbeAlt*.

Der *process*-Block wird für jedes Element der Pipeline einmal aufgerufen. Hier prüft die Funktion den Dateityp und ändert dann die Vordergrundfarbe entsprechend. Anschließend wird das aktuelle Pipeline-Objekt in *\$_* wieder auf die Pipeline gelegt, damit der nachfolgende Befehl es weiterbearbeiten kann.

ACHTUNG Legen Sie *\$_* nicht zurück auf die Pipeline, wird dieses Ergebnis verschluckt. Das ist das Prinzip des Pipeline-Filters *Where-Object*.

Im *end*-Block schließlich finden die Aufräumarbeiten statt, nachdem die Pipeline beendet wurde. Hier restauriert die Funktion die Ausgangsfarben.

ACHTUNG Brechen Sie die Funktion mit `[Strg] + [C]` vorzeitig ab, wird der *end*-Block nicht ausgeführt und die Farben werden nicht auf die Ausgangseinstellung zurückgesetzt.

Ausgaben sortieren

Sie möchten die Ausgaben eines Befehls nach einem bestimmten Kriterium sortieren.

Lösung

Leiten Sie das Ergebnis über die Pipeline an *Sort-Object* weiter. Geben Sie dazu hinter *Sort-Object* an, nach welcher Eigenschaft Sie sortieren wollen. Die folgende Zeile liefert ein alphabetisches Orderlisting sämtlicher *.dll*-Dateien aus dem *System32*-Unterordner des *Windows*-Ordners:

```
PS > dir $env:windir\system32 *.DLL | Sort-Object Name
```

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Windows\system32

Mode	LastWriteTime		Length	Name
----	-----	-----	-----	----
-a---	18.01.2008	22:33	136192	aaclient.dll
-a---	18.01.2008	22:33	2515968	accessibilitycpl.dll
-a---	02.11.2006	08:28	39424	ACCTRES.dll
-a---	02.11.2006	10:46	7680	acledit.dll
-a---	18.01.2008	22:33	127488	acmui.dll
-a---	02.11.2006	10:46	38912	acppage.dll
-a---	02.11.2006	08:11	2048	acprgwiz.dll
(...)				

Möchten Sie in absteigender Reihenfolge sortierten, geben Sie den Parameter *–descending* an. Möchten Sie zwischen Groß- und Kleinschreibung unterscheiden, geben Sie den Parameter *–caseSensitive* an.

Sort-Object kann auch nach berechneten Eigenschaften sortieren, falls keine vorhandene Eigenschaft abbildet, wonach Sie sortieren möchten. Die folgende Zeile sortiert einen Ordnerinhalt nach der Länge der Dateinamen:

```
PS > Dir $env:windir | Sort-Object { $_.Name.Length }
```

Hintergrund

Sort-Object sortiert Objekte nach einer oder mehreren Eigenschaften – sofern die Objekte überhaupt über Eigenschaften verfügen. Primitive Datentypen wie Zahlen oder Texte sortiert *Sort-Object* ohne zusätzliche Angaben:

```
PS > 1,4,2,7,6 | Sort-Object
```

Geben Sie zusätzlich den Parameter *–Unique* an, werden doppelte Resultate aus dem Ergebnis entfernt.

Stellt PowerShell die Ergebnisse in mehreren Spalten oder als Liste dar, wissen Sie, dass es sich um Objekte mit mehreren Eigenschaften handelt. In diesem Fall nennen Sie *Sort-Object* mit dem Parameter *–property* die Eigenschaft(en), nach denen sortiert werden soll. Die folgende Zeile sortiert die DLL-Dateien des Systemordners aufsteigend nach Größe (Eigenschaft *Length*):

```
PS > dir $env:windir\system32 *.dll | Sort-Object Length
```

Sort-Object kann auch nach mehreren Eigenschaften gleichzeitig sortieren. Die nächste Zeile sortiert laufende Prozesse zuerst nach Hersteller (Eigenschaft *Company*) und bei Gleichheit danach nach Name:

```
PS > Get-Process | Sort-Object Company, Name | Select-Object Name, Company
```

HINWEIS

Als Nicht-Administrator dürfen Sie viele Prozesseigenschaften (wie zum Beispiel die Eigenschaft *Company*) nur von solchen Prozessen lesen, die Sie selbst gestartet haben. Bei allen anderen Prozessen ist die Eigenschaft dann leer.

Mit dem Parameter *Descending* drehen Sie die normalerweise aufsteigende Sortierreihenfolge um. Sortieren Sie nach mehreren Eigenschaften, legt der Parameter für alle Eigenschaften eine gemeinsame Sortierreihenfolge fest.

Möchten Sie die Sortierreihenfolge für einzelne Eigenschaften separat festlegen, verwenden Sie ein Hashtable. Die folgenden Zeilen sortieren alle laufenden Prozesse nach Hersteller in aufsteigender Reihenfolge. Der Speicherbedarf der Prozesse wird dagegen in absteigender Reihenfolge angegeben:

```
PS > $kriterium1 = @{expression='Company';Descending=$false}
PS > $kriterium2 = @{expression='VirtualMemorySize';Descending=$true}
PS > Get-Process | Sort-Object $kriterium1, $kriterium2 | Format-Table Name, -->
    Company, VirtualMem*
```

Sort-Object kann auch berechnete Eigenschaften zum Sortieren verwenden. Dazu geben Sie anstelle eines Eigenschaftsnamens einen Skriptblock an, der für jedes Objekt einzeln ausgewertet wird. Innerhalb des Skriptblocks repräsentiert *_* das jeweils untersuchte Objekt. Auf diese Weise konnte *Sort-Object* ein Ordnerlisting nach der Länge des Dateinamens sortieren:

```
PS > Dir $env:windir | Sort-Object { $_.Name.Length }
```

Verzeichnis: C:\Windows

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	06.10.2010 08:28		de
d----	14.07.2009 05:20		PLA
d----	05.02.2011 10:00		Logs
d----	01.11.2009 17:07		Help
d----	10.11.2009 10:19		en-US
d-r-s	12.11.2010 00:45		Fonts
d----	14.07.2009 04:36		system
d----	01.11.2009 09:27		Panther
d----	14.07.2009 07:32		Cursors
d----	14.07.2009 07:32		twain_32
d----	14.07.2009 04:35		SchCache
d----	14.07.2009 05:20		AppCompat
-a---	16.11.2009 11:30	6518	DPINST.LOG

Häufig kann man sich berechnete Eigenschaften aber auch zunutze machen, um den Datentyp zu korrigieren. Die folgende Zeile listet für alle DLL-Dateien aus dem Systemordner den Pfadnamen und die Dateiversion auf:

```
PS > dir $env:windir\system32 -filter *.dll | Select-Object -ExpandProperty VersionInfo | Select-Object FileName, ProductVersion
```

Würden Sie diese Zeile an *Sort-Object* weiterleiten und nach *ProductVersion* sortieren, wäre das Ergebnis nicht korrekt sortiert, denn die Eigenschaft *ProductVersion* verwendet den Datentyp *String* und wird von *Sort-Object* also alphabetisch sortiert. Das bedeutet, dass die hypothetische Versionsnummer »10.1.2.3« kleiner ist als »6.1.2.3«, weil bei der alphabetischen Sortierung zeichenweise vorgegangen wird und »1« kleiner ist als »6«.

Indem Sie *Sort-Object* einen Skriptblock übergeben, in dem die Eigenschaft in den für die Sortierung korrekten Datentyp umgewandelt wird, beheben Sie das Problem:

```
PS > dir $env:windir\system32 -filter *.dll | Select-Object -ExpandProperty VersionInfo | Select-Object FileName, ProductVersion | Sort-Object { try {[System.Version]$_ .ProductVersion } catch { 0 }}
```

Ergebnisse gruppieren

Sie wollen die Ergebnisse eines Befehls nach einem oder mehreren Kriterien gruppieren. Sie möchten zum Beispiel Dienste nach ihrem Status gruppieren.

Lösung

Verwenden Sie das Cmdlet *Group-Object* und geben Sie dahinter das Kriterium an, nach dem Sie gruppieren wollen. Die folgende Zeile gruppiert Dienste nach ihrem Status:

```
PS > Get-Service | Group-Object Status
```

Count	Name	Group
89	Running	{AEADIFilters, AeLookupSvc, Appinfo, AudioEndpointBuilder...}
67	Stopped	{ALG, AppMgmt, clr_optimization_v2.0.50727_32, COMSysApp...}

In der Spalte *Group* werden die gruppierten Objekte aufgeführt. Wenn Sie diese Information nicht benötigen, entfernen Sie sie mit dem Parameter *-noElement*. Dies spart erheblichen Speicherplatz.

Die nächste Zeile gruppiert Prozesse nach ihrem Hersteller:

```
PS > Get-Process | Group-Object Company -noElement
```

Count	Name
1	Adobe Systems Incorporated
11	Microsoft Corporation
1	Huawei Technologies Co., Ltd.
1	Deutsche Post AG

Ebenso lassen sich Ereignislogbucheinträge nach *EntryType* gruppieren:

```
PS > Get-EventLog System | Group-Object EntryType -noElement
```

```
Count Name
-----
48724 Information
 3764 Error
 2985 Warning
```

Oder Sie gruppieren den Inhalt eines Dateiordners nach Dateierweiterung:

```
PS > Dir $env:windir | Group-Object Extension -noElement
```

```
Count Name
-----
    64
     1 .NET
     5 .ini
    13 .exe
     1 .dat
     8 .log
     1 .mif
    (...)

```

Sie können auch nach mehreren Eigenschaften gleichzeitig sortieren. Doppelte Dateien finden Sie in Ihrem Benutzerprofil (in *\$home*) beispielsweise, indem Sie nach *Length* und *CreationTime* gruppieren und alle Gruppen auflisten, die mehr als einmal vorkommen:

```
PS > Dir $home -recurse | Where-Object { $_.Length -gt 1KB } | Group-Object →
    Length, CreationTime | Where-Object { $_.Count -gt 1 } | ForEach-Object →
    { "Möglicherweise doppelt: "; $_.Group | ForEach-Object { '{0} ({1:0.0}KB)' →
    -f $_.FullName, ($_.Length/1KB)} }
```

```
Möglicherweise doppelt:
C:\Users\w7-pc9\video1.wmv (4930,5KB)
C:\Users\w7-pc9\video2.wmv (4930,5KB)
```

```
Möglicherweise doppelt:
C:\Users\w7-pc9\Documents\WindowsPowerShell\Modules\PSImageTools\FilterImagesIn
Directory.ps1 (3,9KB)
C:\Users\w7-pc9\Documents\WindowsPowerShell\Modules\PSImageTools\Example\Filter
ImagesInDirectory.ps1 (3,9KB)
```

HINWEIS

Die identifizierten Dateien müssen nicht doppelt sein. Zwar ist es extrem unwahrscheinlich, dass unterschiedliche Dateien sowohl dieselbe Größe als auch dieselbe Erstellungszeit aufweisen, jedoch nicht unmöglich. Dieses Risiko ist umso größer, je kleiner die Dateien sind, weswegen der Code keine Dateien kleiner als *1KB*

berücksichtigt. Wirklich sicher identifiziert nur ein Dateihash identische Dateiinhalte. Die Erstellung von Hashwerten ist besonders bei größeren Dateien allerdings sehr zeitraubend und ressourcenintensiv und daher nur selten praktikabel.

Hintergrund

Group-Object gruppiert beliebige Objekte auf der Basis einer oder mehrerer Eigenschaften. Als Ergebnis liefert das Cmdlet *Group-Object* Objekte zurück. Jedes *Group-Object* meldet in *Count*, wie viele Objekte darin gruppiert wurden. *Name* meldet, welche Gemeinsamkeit in der zur Gruppierung verwendeten Eigenschaft gefunden wurde. *Group* schließlich enthält die Objekte, die in dieser Gruppe zusammengefasst wurden.

Geht es Ihnen nur darum, Häufigkeiten zu ermitteln, und brauchen Sie also die zugrunde liegenden Objekte nicht, sparen Sie erheblichen Speicherplatz mit dem Parameter *-noElement*. Wird er angegeben, verzichtet *Group-Object* darauf, die zugrunde liegenden Objekte aufzubewahren und in *Group* zurückzuliefern.

```
PS > dir $env:windir | Group-Object Extension -noElement | Sort-Object Name -->
    | ForEach-Object { '{0,30} = kam {1} mal vor' -f $_.Name, $_.Count }
        = kam 64 mal vor
        .bin = kam 1 mal vor
        .dat = kam 1 mal vor
        .dll = kam 2 mal vor
        .exe = kam 13 mal vor
        .ini = kam 5 mal vor
        .log = kam 8 mal vor
        .logs = kam 1 mal vor
        .mif = kam 1 mal vor
        .NET = kam 1 mal vor
        .prx = kam 1 mal vor
        .SCR = kam 1 mal vor
        .txt = kam 2 mal vor
        .xml = kam 2 mal vor
(...)

```

Group-Object kann auch nach berechneten Eigenschaften gruppieren. Dazu geben Sie die Eigenschaft als ausführbaren Skriptblock an. *Group-Object* gruppiert dann nach dem Ergebnis dieses Skriptblocks. Die nächste Zeile teilt Dateien zum Beispiel in zwei Gruppen, nämlich Dateien, die größer sind als 100 KB, und Dateien, die nicht größer sind als 100 KB:

```
PS > dir $env:windir | Group-Object {$_ .Length -gt 100KB}

```

Count	Name	Group
94	False	{AABBCC, addins, AppCompat, AppPatch...}
9	True	{explorer.exe, HelpPane.exe, notepad.exe, ntbtlog.txt...}

Liefert der Gruppierungsausdruck mehr als zwei Ergebnisse, erhalten Sie entsprechend mehr als zwei Gruppen. Die folgende Zeile bildet Gruppen auf der Basis der Anfangsbuchstaben der Dateinamen:

```
PS > dir $env:windir | Group-Object { $_.Name.SubString(0,1).ToUpper() }
```

Count	Name	Group
6	A	{AABBCC, addins, AppCompat, AppPatch...}
4	B	{Boot, Branding, bfsvc.exe, bootstat.dat}
2	C	{CSC, Cursors}
10	D	{de, de-DE, debug, diagnostics...}
(...)		
8	W	{Web, winsxs, win.ini, WindowsUpdate.log...}
2	N	{notepad.exe, ntbtlog.txt}
1	U	{Ultimate.xml}
(...)		

Weil jede Gruppe von einem *GroupInfo*-Objekt repräsentiert wird, das in seiner Eigenschaft *Group* die gruppierten Dateien noch enthält, könnten Sie sich auf diese Weise ein alphabetisch formatiertes Ordnerlisting generieren:

```
PS > dir $env:windir | Group-Object { $_.Name.SubString(0,1).ToUpper() } →
    | ForEach-Object { ''; " - $($_.Name) - "; '-----'; ''; $_.Group }
```

```
- A -
-----

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias
```

Mode	LastWriteTime	Length	Name
d----	19.11.2007 21:57		<DIR> Application Data
d----	13.05.2008 12:43		<DIR> Auswertung
-a---	21.05.2008 09:16	8277	arbeitsblatt.xls
- B -			

d----	30.05.2008 07:16		<DIR> bin
d----	16.11.2007 23:46		<DIR> Bluetooth Software
-a---	31.03.2008 17:03	1355	batch.vbs
-a---	31.03.2008 17:09	1355	batchersatz.vbs
(...)			

Wollen Sie häufiger auf diese Weise Ordnerlistings gruppieren, schreiben Sie sich eine passende Funktion wie zum Beispiel *Group-Alphabetisch*:

```
PS > Function Group-Alphabetisch {
>> $input | Group-Object { $_.Name.SubString(0,1).ToUpper() } | ForEach-Object →
    { ''; " - $($_.Name) - "; '-----'; ''; $_.Group }
>> }
>>
```

Diese Funktion blockiert anders als ein Filter die Pipeline, bis der Vorgängerbefehl sämtliche Ergebnisse geliefert hat. Die Ergebnisse stehen danach in *\$input* zur Verfügung und können nun an *Group-Object* weitergeleitet werden.

```
dir | Group-Alphabetisch
```

TIPP

Das Kriterium für die Gruppenbildung kann, wie Sie gesehen haben, frei berechnet werden. Im folgenden Beispiel werden alle Dateien im Windows-Ordner einer der drei Kategorien klein, mittel und groß zugeordnet:

```
PS > Dir $env:windir | Group-Object { Switch($_.Length) {
>> { $_ -eq $null } { 'Ordner'; continue }
>> { $_ -lt 1KB } { 'klein'; continue }
>> { $_ -lt 1MB } { 'mittel'; continue}
>> default { 'gross' }
>> }}
>>
```

Count	Name	Group
65	Ordner	{AABBCC, addins, AppCompat, AppPatch...}
8	klein	{avisplitter.ini, DirectX.log, iltwain.ini, setuperr.log...}
27	mittel	{bfsvc.exe, bootstat.dat, DPINST.LOG, DtcInstall.log...}
3	gross	{explorer.exe, Reflector.exe, WindowsUpdate.log}

Speichern Sie das Ergebnis in einer Variablen und geben die Parameter *-asHash* und *-asString* an, erhalten Sie damit einen sehr einfachen und übersichtlichen Weg, mit kleinen, mittelgroßen und großen Dateien zu arbeiten:

```
PS > $dateien = Dir $env:windir | Group-Object { switch($_.Length) {
>> { $_ -eq $null } { 'Ordner'; continue }
>> { $_ -lt 1KB } { 'klein'; continue }
>> { $_ -lt 1MB } { 'mittel'; continue}
>> default { 'gross' }
>> }} -asHash -asString
>>
```

```
PS > $dateien.klein
```

```

Verzeichnis: C:\Windows

Mode                LastWriteTime         Length Name
----                -
-a---             14.03.2010      19:00           38 avisplitter.ini
-a---             06.10.2010      08:21          197 DirectX.log
-a---             16.04.2010      14:53           36 iltwain.ini
-a---             14.07.2009      06:51           0  setuperr.log
-a---             10.06.2009     23:08          219 system.ini
-a---             21.06.2010     12:40           16 test.logs
-a---             18.10.2010     16:52           16 test.txt
-a---             06.11.2009     14:06          478 win.ini

PS > $dateien.gross

Verzeichnis: C:\Windows

Mode                LastWriteTime         Length Name
----                -
-a---             31.10.2009      07:34    2870272 explorer.exe
-a---             12.11.2010     16:10    2854328 Reflector.exe
-a---             14.02.2011      08:48    1963673 WindowsUpdate.log

```

Ergebnisse in Text umwandeln

Sie möchten das Ergebnis eines Befehls in Text umwandeln, zum Beispiel, um den Text farbig auszugeben.

Lösung

Verwenden Sie *Out-String*. Die folgende Zeile wandelt das Ergebnis von *Get-Process* in Text um und gibt diesen in rot aus:

```

PS > Get-Process | Out-String -Stream | Write-Host -foreground Red

Handles  NPM(K)  PM(K)  WS(K) VM(M)  CPU(s)    Id ProcessName
-----
      244      23   58092   71480   208    17,21     548 AcroRd32
       76       8    1412     208    27     2656 alg
      211      19    3852     7040    82     1728 AppleMobi...

```

Ohne *Out-String* wären nur die Standardobjekteigenschaften in Text verwandelt worden:

```

PS > Get-Process | Write-Host -foreground Red
System.Diagnostics.Process (AcroRd32)
System.Diagnostics.Process (alg)
System.Diagnostics.Process (AppleMobileDeviceService)
...

```

Hintergrund

Leiten Sie Objekte an Befehle weiter, die eigentlich nur mit Text umgehen können, kommt es zu einer automatischen Umwandlung in Text. Diese Umwandlung wird nicht von PowerShell durchgeführt, sondern durch die *toString()*-Methode von .NET Framework. Objekte werden dabei meist durch ihren Datentypnamen ersetzt:

```
PS > (Get-Process -id $pid).toString()
System.Diagnostics.Process (powershell)
```

Sehr viel intelligenter arbeitet die Textumwandlung von PowerShell. Diese wird häufig automatisch aktiv, zum Beispiel, wenn Sie Objekte in die Konsole ausgeben. Leiten Sie dagegen Objekte an andere Befehle weiter, wird unter Umständen die automatische Konvertierung von .NET Framework aktiv. Damit dies nicht geschieht, wandeln Sie Objekte in solchen Fällen manuell mithilfe des Cmdlets *Out-String* in Text um. Der Parameter *-Stream* sorgt dafür, dass bei mehreren Objekten die einzelnen Objekte auch einzeln in Text umgewandelt werden.

```
PS > $text = Get-Process -id $pid | Out-String -Stream
PS > $text
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
867	35	123052	134532	594	156,66	4084	powershell

```
PS > $text[3]
867      35      123052      134532      594      156,66      4084 powershell
```

Out-String bildet also die Brücke zwischen der objektorientierten internen PowerShell-Welt und der textorientierten Ausgabe-Welt. Dies können Sie sich zunutze machen, um ein einfaches, aber sehr effektives Werkzeug herzustellen. Der folgende Pipeline-Filter namens *grep* wandelt intern die Ergebnisse in Text um und prüft dann, ob ein beliebiges Stichwort darin vorkommt. Falls ja, wird das Objekt durch die Pipeline hindurchgelassen, andernfalls ausgefiltert:

```
PS > Filter grep($stichwort) { if ( ($_ | Out-String) -like "$stichwort*" ) { $_ } }
```

Möchten Sie nun alle laufenden Dienste sehen, genügt:

```
PS > Get-Service | grep running
```

Alle *.exe*-Dateien im Windows-Ordner erhalten Sie nun so:

```
PS > Dir $env:windir | grep .exe
```

Alle Dateien, die am 14. Juli 2010 geändert wurden, finden Sie so:

```
PS > dir $env:windir | grep 14.07.2010
```

Und alle Aliasnamen für *Get-ChildItem* ermittelt diese Zeile:

```
PS > Get-Alias | grep child
```

Sie erhalten also auf diese Weise sehr einfach und leicht Filterergebnisse, denn Sie brauchen nicht länger anzugeben, in welcher Spalte Sie ein bestimmtes Ergebnis erwarten. Dafür allerdings können die Ergebnisse unscharf sein. Zurückgeliefert wird jedes Objekt, das in irgendeiner (sichtbaren) Spalte das Suchwort enthält.

Möchten Sie in sämtlichen Eigenschaften eines Objekts suchen, also auch in den normalerweise verborgenen, ändern Sie den Filter geringfügig und wandeln nicht nur die sichtbaren, sondern auch die unsichtbaren Eigenschaften in Text um, bevor Sie darin nach dem Stichwort suchen:

```
Filter grep($stichwort) { if ( ($_ | Select-Object * | Out-String) →  
-like "$stichwort") { $_ } }
```

Der Filter arbeitet nun langsamer, findet aber mehr. Alle Prozesse der Firma »Microsoft« finden Sie nun so:

```
PS > Get-Process | grep Microsoft
```

Im sichtbaren Ergebnis wird Ihr Suchwort möglicherweise nun gar nicht mehr vorkommen. Das Beispiel hat das Stichwort »Microsoft« beispielsweise in der normalerweise verborgenen Eigenschaft *Company* gefunden:

```
PS > Get-Process | grep Microsoft | Select-Object Name, Company
```

Zusammenfassung

Lediglich neun neue Cmdlets aus Tabelle 4.1 waren nötig, um mithilfe der PowerShell-Pipeline anspruchsvolle Aufgaben in einer einzigen Zeile Code zu lösen.

Aufgabe	Seite
Alle Dateien größer als 1MB auflisten	123
Alle Prozesse mit mehr als 20 Sekunden CPU-Lastung zeigen	123
Alle laufenden Prozesse der Firma »Microsoft« auflisten	123
Alle beendeten Dienste auflisten	123
Nur Dateien oder nur Unterordner eines Ordners anzeigen	123

Tabelle 4.2 Auswahl von Lösungen in diesem Kapitel, die durch die Pipeline ermöglicht wurden

Aufgabe	Seite
Zeilen aus einer Logbuchdatei filtern, die ein bestimmtes Stichwort enthalten	123
Alle Hotfixes finden, die mit einer bestimmten KB-Artikelnummer beginnen	123
Nur die Zeilen des Befehls <i>ipconfig</i> anzeigen, die das Wort »IP« enthalten	124
Aktuelle Videoauflösung anzeigen	129
Die neuesten 10 Errorereignisse ungekürzt in eine Textdatei schreiben	129
Alle Computer aus der Datei <i>serverliste.txt</i> remote per WMI abfragen	130
Alle Computer aus der Datei <i>serverliste.txt</i> herunterfahren	130
Eine Liste mit IP-Adressen generieren	130
Ein IP-Adresssegment anpingen und auf Verfügbarkeit der Rechner prüfen	130
Eine Liste mit PC-Namen generieren	130
Ein IP-Adresssegment per DNS auflösen	131
Sortierte Ordnerlistings erzeugen	133
Doppelt vorkommende Dateien im Dateisystem finden	137
Einen eigenen »grep«-Filter zum leichteren Filtern von Befehlsergebnissen anlegen	142

Tabelle 4.2 Auswahl von Lösungen in diesem Kapitel, die durch die Pipeline ermöglicht wurden (*Fortsetzung*)

Das Prinzip dabei war bei jedem Beispiel identisch: Die Rohergebnisse eines Cmdlets wurden zuerst mit **-Object*-Cmdlets verfeinert, gefiltert und sortiert und anschließend mit *Format-*/Out-**-Cmdlets ausgegeben.

Name	Beschreibung
<i>ForEach-Object</i>	Kurzform: »%«; bearbeitet alle Ergebnisse der Pipeline einzeln und in Echtzeit. Entspricht einer Schleife.
<i>Format-List</i>	Listet die Eigenschaften eines Objekts in Textdarstellung untereinander auf
<i>Format-Table</i>	Listet die Eigenschaften eines Objekts in Textdarstellung nebeneinander auf. Kann die Spaltenbreite optimieren und dafür sorgen, dass Zeilen umgebrochen und nicht abgeschnitten werden.
<i>Group-Object</i>	Gruppert Objekte nach einer Eigenschaft oder einem Kriterium
<i>Out-File</i>	schreibt die Ergebnisse in eine Datei
<i>Out-String</i>	Wandelt ein Objekt in Text um
<i>Select-Object</i>	Beschränkt ein Objekt auf bestimmte gewünschte Eigenschaften und entfernt alle übrigen
<i>Sort-Object</i>	Sortiert das Ergebnis der Pipeline nach einem oder mehreren Kriterien
<i>Where-Object</i>	Kurzform: »?«; lässt nur diejenigen Objekte hindurch, die einem bestimmten Kriterium entsprechen, und filtert die übrigen heraus

Tabelle 4.3 Cmdlets in diesem Kapitel

Kapitel 5

Bedingungen und Schleifen

In diesem Kapitel:

Entscheidungen fällen	146
Mehrere Vergleiche kombinieren	150
Arrayinhalte vergleichen	152
Code ausführen, wenn Bedingung erfüllt ist	153
Elemente der Pipeline einzeln bearbeiten	159
Mit Schleifen Code mehrfach wiederholen	160
Mehrere Ergebnisse bearbeiten	164
Zusammenfassung	168

Die Basis von intelligenten Skripten sind Entscheidungen. Entscheidungen werden durch Bedingungen repräsentiert, und PowerShell kann Code je nach Ergebnis einer Bedingung ausführen oder überspringen. Bedingungen dürfen darüber hinaus dazu eingesetzt werden, um Ergebnisse von Befehlen zu filtern und auf die gewünschten Ergebnisse zu reduzieren.

Schleifen wiederholen bestimmte Codebereiche und sind immer dann wichtig, wenn Sie dieselben Aktionen mehrfach durchführen möchten, also zum Beispiel alle Ergebnisse eines Befehls der Reihe nach ausgewertet werden sollen.

Entscheidungen fällen

Sie möchten Informationen vergleichen und aus dem Ergebnis eine Entscheidung fällen.

Lösung

Verwenden Sie einen Vergleichsoperator und vergleichen Sie damit die Informationen. Vergleichsoperatoren liefern immer das Ergebnis *\$true*, wenn der Vergleich stimmt, oder *\$false*, wenn der Vergleich nicht erfüllt ist.

Sie wollen zum Beispiel prüfen, ob eine Zahl größer ist als eine andere:

```
PS > 50 -gt 40
True
PS > 10.4 -lt 5
False
```

Oder Sie möchten erfahren, ob ein Element in einer Liste von Elementen enthalten ist:

```
PS > 1,4,6,3,2 -contains 6
True
PS > 'Montag', 'Dienstag', 'Mittwoch' -contains 'Mittwoch'
True
```

Sie können auch prüfen, ob ein Stichwort in einem Text vorkommt:

```
PS > 'Hallo Welt' -like '*welt*'
True
PS > 'Hallo Welt' -clike '*welt*'
False
PS > 'Hallo Welt' -clike '*Welt*'
True
PS > 'Hallo Welt' -notlike '*Welt*'
False
```

Genauere Mustervergleiche sind mit regulären Ausdrücken möglich. So könnten Sie mit *-match* prüfen, ob in einem Text eine KB-Artikelnnummer vorkommt:

```
PS > 'Der Artikel hat die KB-Nummer KB123456.' -match 'KB\d{6}'  
True
```

Schließlich können Sie auch überprüfen, ob ein Objekt einem bestimmten Typ entspricht. Die folgende Zeile würde mit `-is` prüfen, ob ein Befehl ein Array zurückliefert, also mehrere Elemente:

```
PS > (Get-Process) -is [Object[]]  
True
```

Und mit dieser Bedingung überprüfen Sie, ob ein Element einem Dateiordner entspricht:

```
PS > (get-item c:\windows) -is [System.IO.DirectoryInfo]  
True
```

Hintergrund

Vergleichsoperatoren vergleichen zwei Informationen miteinander und bilden also die Basis für intelligente Kriterien. So könnten Sie zum Beispiel überprüfen, ob ein Befehl mehr als fünf Ergebnisse liefert:

```
PS> @(Get-Process).Count -gt 5
```

Hierbei wird der Vergleichsoperator `-gt` (»größer als«) eingesetzt. Er überprüft, ob die Information links von ihm größer ist als die Information rechts von ihm. Falls ja, liefert der Operator den Wert *\$true* zurück, sonst *\$false*.

Wie ein Vergleich durchgeführt wird, hängt sehr vom Datentyp ab. Vergleichen Sie zum Beispiel Textinformationen, wird alphabetisch verglichen, bei Zahlen dagegen numerisch:

```
PS > 12 -gt 6  
True  
PS > '12' -gt '6'  
False
```

Arrays verhalten sich wiederum anders. Hier wird der Vergleich auf jedes Element des Arrays einzeln angewendet:

```
PS > 1,2,3,4,5,6 -gt 4  
5  
6  
PS > 'Tobias','Marie','Cofi','Werner' -like '*a*'  
Tobias  
Marie
```

So kann man beispielsweise ermitteln, wie oft eine Zahl in einem Array vorkommt:

```
PS > @(1,4,2,6,3,4,2,3,4,44,12 -eq 4).Count  
3
```

Bei gemischten Datentypen richtet sich PowerShell nach dem Datentyp des Werts, der links vom Operator steht:

```
PS > '12' -gt 6  
False  
PS > 12 -gt '6'  
True
```

Nur bei Zahlenwerten kann es vorkommen, dass ausnahmsweise nicht der Datentyp des links stehenden Werts maßgeblich ist, sondern der des rechts stehenden Datentyps. Hier zählt, welcher Datentyp die Zahlen genauer repräsentiert:

```
PS > 123 -lt 123.4  
True  
PS > 123 -lt "123.4"  
False  
PS > 123 -lt "123.5"  
True
```

Im ersten Fall wendet PowerShell das »Widening« an und wandelt den ersten Wert ins präzisere *Double*-Format um. Danach wird verglichen, ob 123,0 kleiner ist als 123,4. Ergebnis: *True*.

Im zweiten Fall ist der erste Wert vom Typ *Integer* und der zweite vom Typ *String*. Hier richtet sich PowerShell nach dem Datentyp des linken Werts und wandelt den Text in eine Zahl um. Weil dabei gerundet wird, ergibt sich für den rechts stehenden Wert 123, und der Vergleich ergibt *False*.

Im letzten Fall passiert dasselbe, nur wird der Text diesmal aufgerundet und ergibt den Zahlenwert 124. Danach wird verglichen, ob 123 kleiner ist als 124: Ergebnis: *True*.

Über Typumwandlungen kann man den Datentyp explizit vorgeben. Werden beispielsweise Versionsinformationen als Text verglichen, entspricht das Ergebnis wegen des alphabetischen Vergleichs nicht den Erwartungen. Wird der links stehende Datentyp dagegen in den passenden Datentyp umgewandelt, stimmt das Ergebnis:

```
PS > '12.1.5.100' -gt '6.33.2.200'  
False  
PS > [System.Version]'12.1.5.100' -gt '6.33.2.200'  
True
```

PowerShell unterstützt eine Vielzahl von Vergleichsoperatoren. Welcher jeweils richtig ist, hängt davon ab, was genau Sie vergleichen wollen:

Operator	klassisch	Beschreibung	Beispiel	Ergebnis
<i>-eq, -ceq, -ieq</i>	=	Gleichheit	10 <i>-eq</i> 15	<i>\$false</i>
<i>-ne, -cne, -ine</i>	<>	Ungleichheit	10 <i>-ne</i> 15	<i>\$true</i>
<i>-gt, -cgt, -igt</i>	>	Größer	10 <i>-gt</i> 15	<i>\$false</i>
<i>-ge, -cge, -ige</i>	>=	Größer oder gleich	10 <i>-ge</i> 15	<i>\$false</i>
<i>-lt, -clt, -ilt</i>	<	Kleiner	10 <i>-lt</i> 15	<i>\$true</i>
<i>-le, -cle, -ile</i>	<=	Kleiner oder gleich	10 <i>-le</i> 15	<i>\$true</i>
<i>-contains, -ccontains, -icontains</i>		Enthält	1,2,3 <i>-contains</i> 1	<i>\$true</i>
<i>-notcontains, -cnotcontains, -inotcontains</i>		Nicht enthalten	1,2,3 <i>-notcontains</i> 1	<i>\$false</i>
<i>-is</i>		Typgleichheit	<i>\$Array -is [array]</i>	<i>\$true</i>

Tabelle 5.1 Vergleichsoperatoren

Normalerweise unterscheiden alle Vergleichsoperatoren bei Textvergleichen nicht zwischen Groß- und Kleinschreibung (Inhaltsvergleich). Möchten Sie zwischen Groß- und Kleinschreibung unterscheiden (binärer Vergleich), stellen Sie dem Operator ein »c« (für »case-sensitive«) voran:

```
PS > 'kennwort' -eq 'KennWort'
True
PS > 'kennwort' -ceq 'KennWort'
False
PS > 'KennWort' -ceq 'KennWort'
True
```

Sie dürfen Vergleichsoperatoren auch ein »i« voranstellen (für »insensitive«). Der Vergleichsoperator verhält sich dann zwar genauso, als hätten Sie keinen Buchstaben vorangestellt, und unterscheidet bei Textvergleichen also nicht zwischen Groß- und Kleinschreibung. Mit dem vorangestellten »i« können Sie aber explizit zum Ausdruck bringen, dass bei diesem Vergleich die Groß- und Kleinschreibung keine Rolle spielt. Das vorangestellte »i« sorgt also für lesbareren Code.

Es gibt für die meisten Vergleiche komplementäre Vergleichsoperatoren: *-eq* und *-ne* (gleich oder ungleich), *-gt* und *-lt* (größer oder kleiner) oder *-like* und *-notlike* (enthält einen Text oder enthält ihn nicht). Durch die Auswahl des passenden Vergleichsoperators bestimmen Sie maßgeblich, ob ein Vergleich *True* oder *False* ergibt.

Zusätzlich kann das Ergebnis eines Vergleichs mit dem logischen Operator *-not* (Kurzform: »!«) negiert (also umgedreht) werden:

```
PS > $a = 10
PS > $a -gt 5
True
PS > -not ($a -gt 5)
False
PS > !($a -gt 5)
False
```

ACHTUNG Setzen Sie runde Klammern großzügig ein, wenn Sie mit logischen Operatoren wie *–not* arbeiten! Logische Operatoren sind immer am Resultat eines Vergleichs interessiert, nicht am Vergleich selbst. Der Vergleich gehört deshalb in runde Klammern. Runde Klammern sind Unterausdrücke, die zuerst für sich ausgeführt werden und danach nur das Ergebnis zurückliefern. Wenn Sie die Klammern vergessen, erhalten Sie schnell sonderbare Resultate:

```
PS > $a = 10
PS > $a -gt 10
False
PS > -not $a -gt 10
False
PS > -not ($a -gt 10)
True
```

Ohne Klammern würde *–not* den unmittelbar folgenden Ausdruck auswerten, also *\$a*. Weil *\$a* weder *\$true* noch *\$false* ist, würde PowerShell *\$a* in einen booleschen Wert umwandeln. Das Ergebnis wäre erstaunlicherweise *\$true*:

```
PS > [bool]$a
True
```

Der Grund: Alle Zahlenwerte ungleich 0 werden zu *\$true*, nur der Zahlenwert 0 wird zu *\$false* konvertiert:

```
PS > $a = 0
PS > [bool]$a
False
```

Was Sie in Wirklichkeit auswerten wollten, war das Ergebnis des Vergleichs *\$a –gt 10*. Damit nur dieses Ergebnis an *–not* geliefert wird, muss der Ausdruck in runde Klammern gestellt werden. Er wird dann zuerst von PowerShell ausgewertet und nur sein Ergebnis an *–not* weitergereicht.

Weitergehende Hilfe zu Vergleichsoperatoren rufen Sie mit *Help* ab:

```
PS > Help comparison
```

Mehrere Vergleiche kombinieren

Sie möchten mehrere Vergleiche (Fragestellungen) kombinieren und wissen, ob alle davon oder ob mindestens einer davon zutreffen.

Lösung

Setzen Sie logische Operatoren ein. Möchten Sie zum Beispiel wissen, ob alle Vergleiche zutreffen, verwenden Sie *–and*:

```
PS > (12 -gt 5) -and ("Hallo Welt" -like '*welt*')
True
```

Wollen Sie hingegen wissen, ob mindestens einer davon erfüllt ist, setzen Sie *-or* ein:

```
PS > (12 -lt 5) -or ("Hallo Welt" -like '*welt*')
True
```

Darf nur genau einer der beiden Vergleiche erfüllt sein, ist *-xor* (»entweder/oder«) richtig:

```
PS > (12 -gt 5) -xor ("Hallo Welt" -like '*welt*')
False
```

Hintergrund

Weil jeder Vergleichsoperator immer entweder *True* oder *False* liefert, können Sie mehrere Vergleiche mit logischen Operatoren kombinieren. Möchten Sie zum Beispiel eine Bedingung erstellen, die aus zwei Fragestellungen besteht, verknüpfen Sie das Ergebnis der beiden Einzelvergleiche mit dem logischen Operator *-and*. Die folgende Bedingung ergäbe nur *True*, wenn beide Teilvergleiche *True* ergeben:

```
( ($alter -ge 18) -and ($geschlecht -eq "m") )
```

Im Kopf eines Disco-Türstehers könnte also unter Umständen die folgende PowerShell-Bedingung ablaufen:

```
( ($alter -ge 18) -and ($geschlecht -eq "m") ) -or ($geschlecht -eq "w")
```

Achten Sie bei logischen Operatoren darauf, dass diese nur *True* und *False* verknüpfen. Stellen Sie also einzelne Vergleiche in runde Klammern, weil Sie nur die Ergebnisse dieser Vergleiche verknüpfen wollen und nicht etwa die Vergleiche selbst.

Operator	Beschreibung	Linker Wert	Rechter Wert	Ergebnis
<i>-and</i>	Beide Bedingungen müssen erfüllt sein	<i>True</i> <i>False</i> <i>False</i> <i>True</i>	<i>False</i> <i>True</i> <i>False</i> <i>True</i>	<i>False</i> <i>False</i> <i>False</i> <i>True</i>
<i>-or</i>	Eine der beiden Bedingungen muss mindestens erfüllt sein	<i>True</i> <i>False</i> <i>False</i> <i>True</i>	<i>False</i> <i>True</i> <i>False</i> <i>True</i>	<i>True</i> <i>True</i> <i>False</i> <i>True</i>

Tabelle 5.2 Logische Operatoren

Operator	Beschreibung	Linker Wert	Rechter Wert	Ergebnis
<i>-xor</i>	Die eine oder die andere Bedingung muss erfüllt sein, aber nicht beide	<i>True</i> <i>False</i> <i>False</i> <i>True</i>	<i>True</i> <i>False</i> <i>True</i> <i>False</i>	<i>False</i> <i>False</i> <i>True</i> <i>True</i>
<i>-not</i>	Kehrt das Ergebnis um	(entfällt)	<i>True</i> <i>False</i>	<i>False</i> <i>True</i>

Tabelle 5.2 Logische Operatoren (*Fortsetzung*)

Arrayinhalte vergleichen

Sie möchten herausfinden, ob ein Array ein bestimmtes Element enthält, oder Sie möchten wissen, ob eine Information einer Liste mit Vorgabewerten entspricht.

Lösung

Verwenden Sie die Operatoren *-contains* und *-notcontains*. Die folgende Bedingung prüft, ob ein Array den Wert 10 enthält:

```
PS > $Array = 1..100
PS > $Array -contains 10
True
PS > $Array -notcontains 10
False
```

Vergleiche sind mit beliebigen Datentypen möglich. Sie könnten so zum Beispiel auch herausfinden, ob ein bestimmter Name in einer Liste vorkommt:

```
PS > $Array = 'Server1','PC1','PC2'
PS > $Array -contains 'PC1'
True
```

Hintergrund

Die Operatoren *-contains* und *-notcontains* können helfen, Elemente in Listen zu finden. Eine Serverliste ließe sich zum Beispiel so anlegen:

```
PS > 'Server1','PC1','PC2' | Out-File $home\serverliste.txt
```

Wollen Sie die Systeme später bearbeiten, lesen Sie die Liste ein:

```
PS > $liste = Get-Content $home\serverliste.txt
```


Das Ergebnis ist ein Array, und mit *-contains* prüfen Sie, ob ein bestimmtes Element darin enthalten ist:

```
PS > $liste -contains 'PC1'  
True  
PS > $liste -contains 'PC10'  
False
```

Auch komplexere Aufgaben lassen sich mit *-contains* lösen. Wollen Sie beispielsweise Dateien finden, die mehrere verschiedene Dateierweiterungen aufweisen können, legen Sie ein Array mit den erwünschten Dateierweiterungen an und prüfen dann für jede Datei, ob die Dateierweiterung im Array vorkommt:

```
PS > Dir $env:windir | Where-Object { '.exe', '.log', '.ini' -contains $_.Extension }
```

Dieselbe Aufgabe hätten Sie alternativ auch mit *-match* und einem regulären Ausdruck lösen können:

```
PS > Dir $env:windir | Where-Object { $_.Extension -match '\.exe|\.log|\.ini' }
```

Code ausführen, wenn Bedingung erfüllt ist

Sie möchten Code nur dann ausführen, wenn bestimmte Voraussetzungen erfüllt sind.

Lösung

Formulieren Sie zuerst das Entscheidungskriterium mithilfe von Vergleichsoperatoren und gegebenenfalls logischen Operatoren. Sie dürfen auch Cmdlets einsetzen, die als Ergebnis *\$true* oder *\$false* zurückliefern.

Sie benötigen also zuerst einen Ausdruck, der *\$true* oder *\$false* zurückliefert. Diesen können Sie dann mit den drei in PowerShell üblichen Bedingungen verwenden, um Anweisungen abhängig vom Ergebnis des Vergleichs auszuführen. Die drei Bedingungen sind *If*, *Switch* und das Cmdlet *Where-Object*.

Möchten Sie einen Ordner anlegen, falls er noch nicht existiert, verwenden Sie als Kriterium das Cmdlet *Test-Path*. Es liefert *\$false* zurück, wenn der angegebene Ordner noch nicht existiert. Für solche Zwecke eignet sich *If* am besten:

```
If ( (Test-Path c:\neuerordner) -eq $false) {  
    New-Item -Path c:\neuerordner -ItemType Directory | Out-Null  
    Write-Warning "Ordner angelegt"  
}
```

Möchten Sie eine Information gegen verschiedene mögliche Informationen testen, setzen Sie *Switch* ein:

```
PS > Switch ( 1 ) {
>> 1 { "eins" }
>> 2 { "zwei" }
>> 3 { "drei" }
>> 4 { "vier" }
>> 5 { "fünf" }
>> }
>>

eins
```

Switch akzeptiert eine Vielzahl von Eingaben, unter anderem auch Arrays:

```
PS > Switch ( 1,5,2,4,3,1 ) {
>> 1 { "eins" }
>> 2 { "zwei" }
>> 3 { "drei" }
>> 4 { "vier" }
>> 5 { "fünf" }
>> }
>>

eins
fünf
zwei
vier
drei
eins
```

Switch eignet sich ideal, um Rückgabewerte in Klartext umzuwandeln. Die folgende Anweisung liest von der WMI (Windows Management Instrumentation) den Typ des Windows-Betriebssystems und wandelt die Codeziffer in lesbaren Text um:

```
PS > $ID = Get-WmiObject Win32_OperatingSystem | Select-Object -expandProperty →
    OperatingSystemSKU
PS > Switch ($ID) {
>> 0      {'Unbekannt' }
>> 1      {'Ultimate' }
>> 2      {'Home Basic' }
>> 3      {'Home Basic Premium' }
>> 4      {'Enterprise' }
>> 5      {'Home Basic N' }
>> 6      {'Business'}
>> 7      {'Standard Server' }
>> 8      {'Datacenter Server' }
>> 9      {'Small Business Server' }
>> 10     {'Enterprise Server' }
>> 11     {'Starter' }
```

```
>> 12      {'Datacenter Server Core' }
>> 13      {'Standard Server Core' }
>> 14      {'Enterprise Server Core' }
>> 15      {'Enterprise Server Itanium' }
>> 16      {'Business N' }
>> 17      {'Web Server' }
>> 18      {'Cluster Server' }
>> 19      {'Home Server' }
>> 20      {'Storage Express Server' }
>> 21      {'Storage Standard Server' }
>> 22      {'Storage Workgroup Server' }
>> 23      {'Storage Enterprise Server' }
>> 24      {'Small Business Server' }
>> 25      {'Small Business Server Premium Edition'}
>> default {'ID $ID ist unbekannt.' }
>> }
>>
```

Mit *Where-Object* schließlich filtern Sie unerwünschte Ergebnisse eines Befehls aus, indem Sie eine Bedingung formulieren, die für alle Ergebnisse erfüllt sein muss. Die folgende Zeile liefert nur Dateien aus dem Windows-Ordner, die mindestens 1 Mbyte groß sind:

```
PS > Get-ChildItem $env:windir | Where-Object { $_.Length -gt 1MB }
```

Hintergrund

Soll Code konditionell ausgeführt werden, also nur dann, wenn eine bestimmte Bedingung erfüllt ist, setzen Sie diesen Code in geschweifte Klammern und bilden so einen Skriptblock. Mit den Anweisungen *If* oder *Switch* beauftragen Sie dann PowerShell, diesen Skriptblock nur auszuführen, wenn die Bedingung erfüllt ist.

Bei *If* wird die Bedingung in runden Klammern angegeben. Der Skriptblock dahinter wird nur dann ausgeführt, wenn die Bedingung den Wert *\$true* ergibt. Die folgende Zeile führt den Skriptblock nur aus, wenn die Variable *\$kennwort* den Inhalt »geheim« enthält:

```
PS > $kennwort = Read-Host "Kennwort"
PS > If ($kennwort -eq 'geheim') { 'Das Kennwort war in Ordnung' }
```

Fügen Sie einen optionalen *Else*-Block hinzu, wenn Sie einen Skriptblock ausführen wollen, falls die Bedingung nicht erfüllt war:

```
PS > $kennwort = Read-Host "Kennwort"
PS > If ($kennwort -eq 'geheim') { 'Das Kennwort war in Ordnung' } Else →
    { 'Das Kennwort war nicht korrekt' }
```

Mit der optionalen *ElseIf*-Anweisung lassen sich weitere Bedingungen überprüfen. *ElseIf* muss hinter *If*, aber vor *Else* stehen und kann auch mehrfach eingesetzt werden.

Ausgeführt wird stets der Skriptblock hinter der ersten zutreffenden Bedingung. Trifft keine Bedingung zu, wird der Skriptblock hinter *Else* ausgeführt. Die *If*-Struktur führt also maximal einen der angegebenen Skriptblöcke aus.

```
PS > [Int]$alter = Read-Host "Ihr Alter"
PS > If ($alter -lt 12) {
>> 'Sie sind ein Kind'
>> } ElseIf ($alter -lt 18) {
>> 'Sie sind ein Jugendlicher'
>> } Else{
>> 'Sie sind erwachsen'
>> }
>>
```

Wollen Sie mehrere Vergleiche gegen denselben Wert durchführen, ist *Switch* übersichtlicher als *If*. Dabei geben Sie hinter *Switch* in runden Klammern den Ausgangswert an. Innerhalb der *Switch*-Struktur geben Sie dann die Vergleichswerte an. *Switch* prüft in diesem Fall auf Gleichheit. Entspricht der Vergleichswert also genau dem in runden Klammern angegebenen Ausgangswert, wird der dahinterstehende Skriptblock ausgeführt.

```
PS > $land = Read-Host 'Aus welchem Land stammen Sie'
Aus welchem Land stammen Sie: USA
PS > Switch ($land) {
>> 'Deutschland' {'Herzlich willkommen!'}
>> 'England' {'Welcome to the UK!'}
>> 'USA' {'Welcome to the US!'}
>> default {'Leider spreche ich Ihre Sprache nicht ...' }
>> }
>>
Welcome to the US!
```

Alternativ dürfen Sie anstelle der festen Vergleichswerte auch Skriptblöcke verwenden. Darin repräsentiert `$_` den Ausgangswert. So können Sie den Ausgangswert auch gegen Wertebereiche testen und dafür beliebige Vergleichsoperatoren einsetzen:

```
PS > [Int]$alter = Read-Host "Ihr Alter"
PS > Switch ($alter) {
>> { $_ -lt 12 } { 'Sie sind ein Kind' }
>> { $_ -lt 18 } { 'Sie sind ein Jugendlicher' }
>> default { 'Sie sind erwachsen' }
>> }
>>
```

Im Gegensatz zu *If* führt *Switch* also sämtliche Skriptblöcke mit einer gültigen Bedingung aus.

```

PS > [int]$zahl = Read-Host 'Geben Sie eine Zahl ein'
Geben Sie eine Zahl ein: 5
PS > Switch ($zahl) {
>> 1 { 'Sie haben die "1" eingegeben' }
>> 5 { 'Sie haben die "5" eingegeben' }
>> { $_ -lt 0 } { 'Sie haben eine negative Zahl eingegeben' }
>> { $_ -ge 0 } { 'Sie haben eine positive Zahl eingegeben' }
>> { ($_ -ge 5) -and ($_ -lt 10) } { 'Zahl liegt zwischen 5 und 9' }
>> default { 'Kein Kriterium traf zu' }
>> }
>>
Sie haben die "5" eingegeben
Sie haben eine positive Zahl eingegeben
Zahl liegt zwischen 5 und 9

```

Wollen Sie das nicht, geben Sie im Skriptblock die Anweisung *Break*. Jetzt werden alle noch folgenden Prüfungen übersprungen. Das nächste Beispiel führt deshalb immer nur die erste zutreffende Bedingung aus:

```

PS > [int]$zahl = Read-Host 'Geben Sie eine Zahl ein'
Geben Sie eine Zahl ein: 5
PS > Switch ($zahl) {
>> 1 { 'Sie haben die "1" eingegeben'; break }
>> 5 { 'Sie haben die "5" eingegeben'; break }
>> { $_ -lt 0 } { 'Sie haben eine negative Zahl eingegeben'; break }
>> { $_ -ge 0 } { 'Sie haben eine positive Zahl eingegeben'; break }
>> { ($_ -ge 5) -and ($_ -lt 10) } { 'Zahl liegt zwischen 5 und 9'; break }
>> default { 'Kein Kriterium traf zu' }
>> }
>>
Sie haben die "5" eingegeben

```

Vergleichen Sie Texte, stehen Ihnen zusätzlich die folgenden Parameter zur Verfügung:

Switch-Parameter	Beschreibung
-exact	Beim Vergleich von Zeichenfolgen wird Groß- und Kleinschreibung unterschieden
-regex	Vergleiche werden mit regulären Ausdrücken durchgeführt
-wildcard	Platzhalterzeichen wie »*«, »?« und »[a-z]« dürfen im Vergleich verwendet werden

Tabelle 5.3 Switch-Parameter für Textvergleiche

Das folgende Beispiel verwendet den Parameter *-wildcard*:

```
PS > $land = Read-Host 'Aus welchem Land stammen Sie'
Aus welchem Land stammen Sie: deutsches Land
PS > Switch -wildcard ($land) {
>> 'Deu*' { 'Deutschland' }
>> 'US*' { 'USA' }
>> 'Eng*' { 'England' }
>> }
>>
Deutschland
```

Noch flexibler sind reguläre Ausdrücke, die Muster noch genauer beschreiben können:

```
PS > $land = Read-Host 'Aus welchem Land stammen Sie'
Aus welchem Land stammen Sie: Österreich
PS > Switch -regex ($land) {
>> '^(^Deu|^Öst|^Schw)' { 'deutscher Sprachraum' }
>> '^(^Eng|^US)' { 'englischer Sprachraum' }
>> default { 'unbekannter Sprachraum' }
>> }
>>
deutscher Sprachraum
```

Das Ergebnis des Mustervergleichs steht Ihnen wie bei regulären Ausdrücken üblich in der Variablen *\$matches* zur Verfügung. Im folgenden Beispiel sucht PowerShell nach bestimmten Sprachmerkmalen, die an beliebiger Stelle im Text vorkommen dürfen. Gibt der Anwender auf die Frage, aus welchem Land er stammt, zum Beispiel »Ich komme aus dem schönen Österreich« an, findet der reguläre Ausdruck den Schlüsselbegriff »Öst«. Die Angabe »\b« sorgt dafür, dass dieser Schlüsselbegriff am Anfang eines Worts stehen muss und nicht irgendwo darin vorkommen darf. Anschließend erweitert der reguläre Ausdruck das gefundene Wort mit »[a-z]*« um alle folgenden Buchstaben.

Korrekterweise meldet PowerShell trotz der etwas blumigen Benutzereingabe nüchtern einen »deutschen Sprachraum« und gibt in *\$matches[0]* das Land an, das der Benutzer tatsächlich eingegeben hat.

```
PS > $land = Read-Host 'Aus welchem Land stammen Sie'
Aus welchem Land stammen Sie: Ich komme aus dem schönen Österreich
PS > switch -regex ($land) {
>> '(\b(Deu|Öst|Schw)[a-z]*)' { "deutscher Sprachraum: $($matches[0])" }
>> '(\b(Eng|US)[a-z]*)' { "englischer Sprachraum: $($matches[0])" }
>> default { "unbekannter Sprachraum: $($matches[0])" }
>> }
>>
deutscher Sprachraum: Österreich
```

Where-Object ist im Gegensatz zu *If* und *Switch* kein Schlüsselwort, sondern ein Cmdlet, das in der PowerShell-Pipeline eingesetzt wird. Es empfängt die Ergebnisse eines vorangegangenen

Befehls und leitet die Ergebnisse nur dann an den nächsten Befehl weiter, wenn die angegebene Bedingung erfüllt ist. Die Entscheidung, ob ein Ergebnis weitergeleitet oder ausgefiltert wird, legen Sie mit einem Skriptblock fest. Dieser wird für jedes Objekt, das *Where-Object* empfängt, erneut ausgeführt. Innerhalb des Skriptblocks repräsentiert `$_` das jeweils zu untersuchende Objekt.

Die folgende Zeile liefert nur Dienste, die augenblicklich laufen:

```
PS > Get-Service | Where-Object -filterScript { $_.Status -eq 'Running' }
```

Speichert man das Filterskript in einer separaten Variablen, würde der Aufruf folgendermaßen aussehen:

```
PS > $filter = { $_.Status -eq 'Running' }  
PS > Get-Service | Where-Object -filterScript $filter
```

Jetzt wird verständlich, wie *Where-Object* intern aufgebaut ist und wo die Bedingung ins Spiel kommt. Der folgende Code veranschaulicht, was bei *Where-Object* hinter den Kulissen tatsächlich geschieht:

```
PS > $filter = { $_.Status -eq 'Running' }  
PS > Get-Service | ForEach-Object -process { If ( & $filter) { $_ } }
```

Where-Object ist also lediglich eine praktische Kurzform einer *ForEach-Object*-Schleife mit einer *If*-Bedingung darin.

Elemente der Pipeline einzeln bearbeiten

Sie möchten die Ergebnisse eines Befehls in der Pipeline einzeln Objekt für Objekt bearbeiten.

Lösung

Verwenden Sie die *ForEach-Object*-Schleife (Kurzform `»%«`):

```
PS > 1..5 | ForEach-Object { "Bearbeite gerade $_" }  
Bearbeite gerade 1  
Bearbeite gerade 2  
Bearbeite gerade 3  
Bearbeite gerade 4  
Bearbeite gerade 5
```

Mit *ForEach-Object* lassen sich leicht Listen erstellen, indem Sie Zahlenfolgen beispielsweise in IP-Adressen oder PC-Namen umformen:

```
PS > 1..255 | ForEach-Object { "192.168.2.$ " }
PS > 1..100 | ForEach-Object { 'PC_{0:000}' -f $_ }
```

Auf gleiche Weise bearbeiten Sie auch komplexe Objekte. Empfängt *ForEach-Object* Daten, die aus mehreren Einzelinformationen (Eigenschaften) bestehen, geben Sie hinter `$_` einen Punkt und dann den Namen der Eigenschaft an, auf die Sie zugreifen möchten:

```
PS > Get-WmiObject Win32_UserAccount | % { '{0,30}={1,-30}' -f $_.Name, $_.SID }
Administrator=S-1-5-21-2613171836-1965730769-3820153312-500
ASPNET=S-1-5-21-2613171836-1965730769-3820153312-1001
Fred=S-1-5-21-2613171836-1965730769-3820153312-1016
Gast=S-1-5-21-2613171836-1965730769-3820153312-501
Karlheinz=S-1-5-21-2613171836-1965730769-3820153312-1021
KunoKillerbach=S-1-5-21-2613171836-1965730769-3820153312-1018
Kurt=S-1-5-21-2613171836-1965730769-3820153312-1022
Maria=S-1-5-21-2613171836-1965730769-3820153312-1017
(...)
```

Hintergrund

Die Pipeline übergibt die Ergebnisse des vorangegangenen Befehls in Echtzeit an den nachfolgenden Befehl. *ForEach-Object* liest diese Ergebnisse Objekt für Objekt und stellt die Objekte in der Variablen `$_` zur Verfügung. Sie können so die Objekte innerhalb der Pipeline bearbeiten.

ForEach-Object unterstützt entweder einen oder drei Skriptblöcke. Geben Sie drei an, wird der erste vor dem Start ausgeführt, der zweite für jedes Pipelineobjekt einmal und der dritte am Ende:

```
PS > ping.exe 10.10.10.10 -n 1 | % {'führe PING durch...'} →
    { If ($? -like '*pakete*') { $_ } } {'Erledigt.'}
führe PING durch...
Pakete: Gesendet = 1, Empfangen = 0, Verloren = 1 (100% Verlust),
Erledigt.
```

Code mit Schleifen mehrfach wiederholen

Sie möchten einen bestimmten Code mehrfach wiederholen, zum Beispiel, weil Sie den Anwender so lange nach einer Eingabe fragen möchten, bis eine korrekte Eingabe vorgenommen wurde.

Lösung

Verwenden Sie eine fußgesteuerte *Do*-Schleife, wenn Sie nicht wissen, wie oft die Schleife wiederholt werden muss. Die folgende Schleife fragt den Anwender nach seiner Homepage-Webadresse. Die Schleife wird erst dann beendet, wenn eine Webadresse eingegeben wurde, die den Prüfungskriterien entsprach:


```
PS > Do {  
>> $eingabe = Read-Host "Ihre Homepage"  
>> } While ($eingabe -notlike "www.*.*")  
>>  
Ihre Homepage: weiss ich nicht  
Ihre Homepage: www.powershell.de
```

Wissen Sie dagegen bereits von vornherein, wie oft die Schleife laufen soll, setzen Sie *For* ein. Die folgende *For*-Schleife verwendet eine Schrittweite von 300 und liefert Frequenzen, die anschließend über eine .NET-Methode hörbar gemacht werden:

```
PS > For ($f=300; $f -lt 3000; $f+=300) { "$f Hz"; [System.Console] →  
::Beep($f, 300) }
```

Hintergrund

Eine *Do*-Schleife ist von Natur aus eine Endlosschleife und reagiert mit ihrem Abbruchkriterium auf Bedingungen, die bei jedem Schleifendurchlauf neu ausgewertet werden. Am Ende der Beispielschleife hinter *While* steht das Kriterium, das *erfüllt* sein muss, damit die Schleife noch einmal wiederholt wird. Im Beispiel wird mit *-notlike* geprüft, ob die Eingabe dem Muster »www.*.*« entspricht. Ist die Bedingung erfüllt, hat der Anwender keine gültige Webadresse eingegeben und die Schleife fragt erneut nach.

Bei dieser Art der Endlosschleife findet die Überprüfung des Schleifenkriteriums erst am Ende statt. Die Schleife wird deshalb mindestens einmal durchlaufen und »fußgesteuert« genannt. Daneben gibt es »kopfgesteuerte« Schleifen, die noch vor dem ersten Schleifendurchlauf die Prüfung durchführen. Dazu stellen Sie die *While*-Anweisung mit ihrem Kriterium an den Anfang der Schleife (und lassen das nutzlos gewordene *Do* weg):

```
PS > $file = [system.io.file]::OpenText("C:\autoexec.bat")  
PS > While (!$file.EndOfStream) {  
>> $file.ReadLine()  
>> }  
>>  
REM Dummy file for NTVDM  
PS > $file.close()
```

HINWEIS

Dies ist selbstverständlich nur ein konstruiertes Beispiel. Textdateien lesen Sie sehr viel einfacher mit dem Cmdlet *Get-Content*:

```
PS > Get-Content c:\autoexec.bat  
REM Dummy file for NTVDM
```

Ob die Schleife fortgesetzt werden soll oder nicht, können Sie auch über Variablen steuern, denn hinter *While* wird lediglich der Wert *\$true* oder *\$false* erwartet. Er muss nicht das Ergebnis eines

Vergleichs sein. Die folgende Schleife fragt erneut nach einer Webseitenadresse, gibt diesmal aber Hinweistexte aus, wenn der Anwender keine gültige Angabe gemacht hat:

```
PS > Do {
>> $eingabe = Read-Host "Ihre Homepage"
>> if ($eingabe -like "www.*.*") {
>> $nachfragen = $false
>> } else {
>> Write-Host -fore "Red" "Eine gültige Webadresse bitte!"
>> $nachfragen = $true
>> }
>> } While ($nachfragen)
>>
Ihre Homepage: sag ich nicht
Eine gültige Webadresse bitte!
Ihre Homepage: na gut
Eine gültige Webadresse bitte!
Ihre Homepage: www.powershell.de
```

Das Beispiel lässt sich weiter vereinfachen, indem Sie absichtlich eine Endlosschleife starten. Dazu wird als Fortsetzungskriterium der Wert *\$true* fest hinterlegt, sodass die Schleife endlos wiederholt wird. Mit der Anweisung *break* kann die Schleife manuell abgebrochen werden:

```
PS > While ($true) {
>> $eingabe = Read-Host "Ihre Homepage"
>> if ($eingabe -like "www.*.*") {
>> break
>> } else {
>> Write-Host -fore "Red" "Eine gültige Webadresse bitte!"
>> }
>> }
>>
Ihre Homepage: ???
Eine gültige Webadresse bitte!
Ihre Homepage: www.powershell.de
```

Wissen Sie genau, wie oft Sie einen bestimmten Codeteil wiederholen wollen, setzen Sie die *For*-Schleife ein. So könnten Sie eine ASCII-Codetabelle erstellen:

```
PS > for ($x=32; $x -le 255; $x++ ) { '{0}: {1}' -f $x, [char]$x }
32:
33: !
34: "
35: #
36: $
(...)
```

Die *For*-Schleife ist nur eine Sonderform der *While*-Schleife, wertet allerdings im Gegensatz zu ihr insgesamt drei Ausdrücke aus:

- **Initialisierung** Der erste Ausdruck wird ausgewertet, wenn die Schleife beginnt

- **Fortsetzungskriterium** Der zweite Ausdruck wird vor jedem Durchlauf ausgewertet. Er entspricht im Grunde dem Fortsetzungskriterium der *While*-Schleife. Ist dieser Ausdruck *\$true*, wird die Schleife wiederholt.
- **Schrittweite** Der dritte Ausdruck wird ebenfalls bei jedem Durchlauf ausgewertet, spielt aber für die Wiederholung keine Rolle. Achtung: Dieser Ausdruck kann keine Ausgaben generieren. Sie können in diesem Ausdruck also keinen sichtbaren Text ausgeben.

Eine *For*-Schleife kann beispielsweise zu einer *While*-Schleife werden, wenn Sie den ersten und den dritten Ausdruck ignorieren und nur den zweiten Ausdruck – das Fortsetzungskriterium – verwenden. Im folgenden Beispiel verhalten sich *For* und *While* gleich:

```
PS > $i = 0
PS > While ($i -lt 5) {
>> $i++
>> $i
>> }
>>
1
2
3
4
5
PS > $i = 0
PS > For (;$i -lt 5;)
>> {
>> $i++
>> $i
>> }
>>
1
2
3
4
5
```

Während in diesem Beispiel die *While*-Schleife vorzuziehen ist, lassen sich mit *For* auch ungewöhnlichere Schleifen realisieren. Die Benutzerabfrage lässt sich so auf wenige Zeilen reduzieren:

```
PS > For ($eingabe=""; !($eingabe -like "www.*.*"); →
    $eingabe = Read-Host "Ihre Homepage") {
>> Write-Host -fore "Red" "Geben Sie eine gültige Webadresse ein!"
>> }
>>
Geben Sie eine gültige Webadresse ein!
Ihre Homepage: sag ich nicht
Geben Sie eine gültige Webadresse ein!
Ihre Homepage: www.powershell.de
```

Im ersten Ausdruck wird die Variable *\$eingabe* auf einen leeren Text gesetzt. Der zweite Ausdruck prüft, ob in *\$eingabe* eine gültige Webadresse steht, und falls ja, wird das Ergebnis mit »!«

umgedreht, ist also *\$true*, wenn keine gültige Webadresse in *\$eingabe* steht. In diesem Fall wird die Schleife also wiederholt. Im dritten Ausdruck wird der Anwender nach einer Webadresse gefragt. Innerhalb der Schleife braucht eigentlich gar nichts mehr zu stehen. Im Beispiel wird ein Hinweistext ausgegeben. Und auch das zeilenweise Auslesen einer Textdatei kann von einer *For*-Schleife mit weniger Code erledigt werden:

```
PS > For ($file = [system.io.file]::OpenText("C:\autoexec.bat"); →
    !($file.EndOfStream); $zeile = $file.ReadLine())
>> {
>> $zeile
>> }
>> $file.close()
>>
REM Dummy file for NTVDM
```

ACHTUNG Der dritte Ausdruck der *For*-Schleife wird vor jedem Schleifendurchlauf ausgeführt. Im Beispiel wird hier die aktuelle Zeile aus der Textdatei gelesen. Dieser dritte Ausdruck wird immer unsichtbar ausgeführt. Sie können darin also keine Textausgaben vornehmen. Deshalb wird der Inhalt der Zeile innerhalb der Schleife ausgegeben.

Mehrere Ergebnisse bearbeiten

Sie möchten den Inhalt eines Arrays einzeln der Reihe nach auswerten. Sie wollen zum Beispiel alle Ergebnisse eines Befehls der Reihe nach bearbeiten oder auswerten.

Lösung

Setzen Sie die *ForEach*-Schleife ein:

```
PS > $Array = 1..5
PS > Foreach ($element in $Array) { "bearbeite $element" }
bearbeite 1
bearbeite 2
bearbeite 3
bearbeite 4
bearbeite 5
```

Alternativ verwenden Sie die Pipeline und *ForEach-Object* (Kurzform »%«):

```
PS > $Array = 1..5
PS > $Array | ForEach-Object { "bearbeite $_" }
bearbeite 1
bearbeite 2
bearbeite 3
bearbeite 4
bearbeite 5
```

Die Anweisung *Foreach* ist erheblich schneller als die Pipeline-Variante, die das Cmdlet *ForEach-Object* einsetzt. Dafür ist der Speicherbedarf der Pipeline-Variante niedriger, weil innerhalb der Pipeline nur jeweils ein Objekt im Speicher bearbeitet wird.

Hintergrund

Foreach und *ForEach-Object* scheinen auf den ersten Blick sehr ähnlich zu funktionieren. Während *ForEach-Object* seine Eingaben allerdings aus der Pipeline bezieht und das aktuelle Objekt innerhalb der Schleife in der Variablen `$_` bereitstellt, geben Sie bei *Foreach* den Container mit den Daten selbst an und legen auch selbst fest, unter welchem Namen das aktuelle Element in der Schleife angesprochen wird:

```
PS > dir C:\ | ForEach-Object { $_.name }
PS > Foreach ($element in dir C:\) { $element.name }
```

Welche Schleife Sie einsetzen, ist aber keine reine Geschmackssache. Die Pipeline-Konstruktion mit *ForEach-Object* arbeitet in Echtzeit und benötigt sehr wenig Speicherplatz. Untersuchen Sie beispielsweise rekursiv umfangreiche Ordnerstrukturen, erhalten Sie so bereits erste Ergebnisse, noch während der Ursprungsbefehl die Daten beschafft:

```
PS > dir C:\ -recurse -ErrorAction SilentlyContinue | ForEach-Object { $_.name }
```

Foreach dagegen ruft zuerst sämtliche Ergebnisse ab, unterstützt also keinen Echtzeitmodus. Bei langwierigen Operationen sehen Sie also lange Zeit keine Ergebnisse und alle Ergebnisse werden zuerst im Speicher angehäuft, benötigen also sehr viel Speicherplatz:

```
PS > Foreach ($element in dir C:\ -recurse -ErrorAction SilentlyContinue) →
{ $element.name }
```

Dafür ist *Foreach* im Gegensatz zu *ForEach-Object* aber erheblich schneller. Immer, wenn die Ergebnisse, die Sie auswerten wollen, ohnehin bereits komplett vorliegen, zum Beispiel in einer Variablen, ist *Foreach* deshalb die bessere Wahl:

```
PS > $Array = 3,6,"Hallo", 12
PS > (Measure-Command {$Array | ForEach-Object →
    { "Aktuelles Element: $_" }}).totalmilliseconds
1,047
PS > (Measure-Command {Foreach ($element in $Array) →
    {"Aktuelles Element: $element" }}).totalmilliseconds
0,0611
```

Daraus ergeben sich diese Regeln:

- **ForEach-Object** Müssen Sie die Ergebnisse, die die Schleife auswerten soll, erst noch beschaffen, und dauert diese Beschaffung mehr als ein paar Millisekunden, verwenden Sie

ForEach-Object und die Pipeline, damit es zu keinen längeren Wartezeiten kommt und die Ergebnisse sofort in dem Moment bearbeitet werden, wo sie vorliegen

- **Foreach** Haben Sie die Ergebnisse bereits in einer Variablen vorliegen oder geht die Beschaffung blitzschnell, verwenden Sie *Foreach* mit seinem Geschwindigkeitsvorteil, denn *Foreach* spart sich den Overhead der Pipeline

Sowohl für *ForEach-Object* als auch für *Foreach* gibt es einige Besonderheiten zu beachten:

ForEach-Object unterstützt mit *-begin* und *-end* zwei optionale Parameter, denen man weitere Skriptblöcke übergeben kann. So lassen sich vor Beginn der Schleife Initialisierungsarbeiten und nach Abschluss der Schleife Aufräumarbeiten durchführen.

```
PS > 1..10 | ForEach-Object -begin { 'Start' } -process { "bearbeite $_" } -end { 'Ende' }
Start
bearbeite 1
bearbeite 2
(...)
bearbeite 10
Ende
```

Solange die *Foreach*-Schleife läuft, haben Sie mit der Variablen *\$Foreach* Zugriff auf den Schleifenmechanismus und könnten Elemente im Container überspringen oder ein zweites Mal durchlaufen. Die folgende *Foreach*-Schleife gibt zum Beispiel nur ungerade Zahlen aus:

```
PS > Foreach ($i in 1..30) {
>> $i
>> $Foreach.MoveNext() | Out-Null
>> }
>>
1
3
5
(...)
```

Weil der Befehl *MoveNext()* einen nicht benötigten Wert zurückmeldet, wird dieser an *Out-Null* weitergegeben, also vernichtet. Der aktuelle Wert der Schleife wird in *\$Foreach.Current* ausgegeben. Sie hätten also auch schreiben können:

```
PS > Foreach ($i in 1..30) {
>> $Foreach.Current
>> $Foreach.MoveNext() | Out-Null
>> }
>>
```

Nützlich ist das, wenn das Array, das Sie auswerten wollen, Informationen paarweise gruppiert. Dann nämlich könnte die *Foreach*-Schleife zuerst das erste Element lesen, anschließend mit *\$Foreach.MoveNext()* zum nächsten Element gehen und dieses dann mit *\$Foreach.Current* ebenfalls lesen. Im Endeffekt würde die Schleife so bei jedem Durchlauf gleich zwei Elemente verarbeiten, so wie hier, wo die aktuellen Netzwerkdruckerzuordnungen sichtbar gemacht werden:

```
PS > $network = New-Object -comObject WScript.Network
PS > $druckerliste = $network.EnumPrinterConnections()
PS > Foreach ($element in $druckerliste) {
>> [void] $Foreach.MoveNext()
>> "Drucker {0} entspricht {1}" -f $Foreach.Current, $element
>> }
>>
Drucker SnagIt 8 entspricht C:\ProgramData\TechSmith\SnagIt 8\PrinterPortFile
Drucker Samsung CLX-216x Farbe entspricht IP_192.168.2.103
Drucker Microsoft XPS Document Writer entspricht XPSPort:
Drucker HP LaserJet Plain Paper entspricht 192.168.2.250
Drucker Fax entspricht SHRFX:
Drucker DYM0 LabelWriter 400 Turbo entspricht USB001
```

Auch die *Switch*-Konstruktion kann ganz ähnlich wie *Foreach* zum Durchlaufen von Arrays verwendet werden:

```
PS > $Array = 1..5
PS > Switch ($Array)
>> {
>> Default { "Aktuelles Element: $_" }
>> }
>>
Aktuelles Element: 1
Aktuelles Element: 2
Aktuelles Element: 3
Aktuelles Element: 4
Aktuelles Element: 5
```

Die Laufvariable, die bei jedem Schleifendurchlauf das aktuelle Element des Arrays liefert, kann bei *Switch* im Gegensatz zu *Foreach* nicht benannt werden, sondern heißt immer *\$_*. Ansonsten funktioniert der äußere Teil der Schleife genau gleich. Im Inneren der Schleife gibt es einen weiteren Unterschied: Während *Foreach* für jeden Schleifendurchlauf immer denselben Code ausführt, kann *Switch* hier mithilfe von Bedingungen wahlweise unterschiedlichen Code ausführen. Im einfachsten Fall enthält die *Switch*-Schleife nur die *default*-Anweisung. Dahinter steht in geschweiften Klammern der Code, der ausgeführt werden soll.

Wenn Sie also bei jedem Schleifendurchlauf ohnehin genau dieselben Anweisungen ausführen wollen, ist *Foreach* die richtige Wahl. Möchten Sie dagegen die einzelnen Elemente des Arrays je nach Inhalt unterschiedlich behandeln, verwenden Sie besser *Switch*:

```

PS > $Array = 1..5
PS > Switch ($Array)
>> {
>> 1 { "Die Zahl 1" }
>> {$ _ -lt 3} { "$ _ ist kleiner als 3" }
>> {$ _ % 2} { "$ _ ist ungerade" }
>> Default { "$ _ ist gerade" }
>> }
>>
Die Zahl 1
1 ist kleiner als 3
1 ist ungerade
2 ist kleiner als 3
3 ist ungerade
4 ist gerade
5 ist ungerade

```

Zusammenfassung

Bedingungen führen Code nur unter bestimmten Voraussetzungen aus. Um solche Voraussetzungen zu beschreiben, verwenden Sie Vergleichsoperatoren (Tabelle 5.1 auf Seite 149). Sollen mehrere Vergleiche kombiniert werden, lassen sich diese mit logischen Operatoren verknüpfen (Tabelle 5.2 auf Seite 151). Das Ergebnis ist jeweils *\$true* (wenn der Vergleich zutrif) oder *\$false*.

Bedingungen werten das Ergebnis von Vergleichen aus. Empfangen sie ein *\$true*, wird der zugeordnete Skriptblock ausgeführt, sonst nicht. PowerShell kennt die *If*-Bedingung sowie den Sonderfall *Switch*, der mehrere Vergleiche gegen denselben Wert ausführt. Innerhalb der Pipeline kann der Einfachheit halber das Cmdlet *Where-Object* eingesetzt werden. Es greift intern auf *If* zurück, um festzustellen, ob ein Pipeline-Element an den nächsten Befehl der Pipeline gesendet wird oder nicht.

Bedingung	Beschreibung
<i>If</i>	Bedingung, führt Skriptblöcke je nach erfüllter Bedingung aus
<i>Switch</i>	Kombination aus Schleife und Bedingung und Sonderform von <i>If</i>
<i>Where-Object</i>	Cmdlet, das innerhalb der Pipeline eingesetzt wird, um intern mit <i>If</i> eine Bedingung zu formulieren. Trifft sie zu, wird das aktuelle Pipeline-Element an den nächstfolgenden Befehl weitergeleitet, andernfalls herausgefiltert.

Tabelle 5.4 PowerShell-Konstrukte in diesem Kapitel

Schleifen wiederholen Skriptblöcke so lange, bis ein Abbruchkriterium erfüllt ist. Die *Do*-Schleife kann das Kriterium wahlweise am Anfang (»kopfgesteuert«) oder am Ende (»fußgesteuert«) auswerten.

Die *For*-Schleife eignet sich besonders gut dazu, eine vordefinierte Anzahl von Wiederholungen in einer beliebigen Schrittweite durchzuführen, ist also eine Zählschleife. Möchten Sie eine Aufgabe beispielsweise genau zehnmal durchführen, ist diese Schleife ideal.

Foreach wiederholt den Skriptblock für jedes Element in einem Container, zum Beispiel in einem Array. Diese Schleife wird beispielsweise dazu verwendet, um alle Ergebnisse eines Befehls auszuwerten. Die Anzahl der Wiederholungen ergibt sich also aus der Anzahl der Ergebnisse. *Foreach* ist besonders schnell, muss aber alle Ergebnisse gleichzeitig im Speicher halten, was Speicherplatz kosten kann.

Das Cmdlet *ForEach-Object* wird in der Pipeline eingesetzt und bearbeitet alle durch die Pipeline gesendeten Objekte der Reihe nach. Weil in der Pipeline immer nur jeweils ein Objekt zur Zeit bearbeitet wird, ist der Speicherplatzbedarf sehr gering. Dafür kostet der aufwändige Übergabemechanismus der Pipeline Rechenzeit. *ForEach-Object* ist deshalb langsamer als *Foreach*.

Schleife	Beschreibung
<i>Do...While</i>	Schleife mit Prüfung der Fortsetzungsbedingung am Schleifenende
<i>For</i>	Zählschleife
<i>Foreach</i>	Schleife, die alle Elemente eines Containers der Reihe nach bearbeitet. Sehr schnell, aber speicherplatzintensiv, weil der gesamte Containerinhalt im Speicher gehalten wird
<i>While</i>	Schleife mit Prüfung der Fortsetzungsbedingung am Schleifenanfang
<i>ForEach-Object</i>	Pipeline-Schleife, besonders speicherplatzschonend, aber langsamer als <i>Foreach</i>

Tabelle 5.5 PowerShell-Konstrukte in diesem Kapitel

Kapitel 6

Skripts, Funktionen und Fehlerbehandlung

In diesem Kapitel:

Skripts verfassen	172
Skripts starten	174
Skripts automatisch ausführen	176
Skripts in der Pipeline verwenden	177
Rückgabewert eines Skripts festlegen	180
Optionale Parameter definieren	181
Zwingend erforderliche Parameter definieren	183
Switch-Parameter definieren	184
Parameter validieren	186
Mehrere Werte pro Parameter übergeben	187
Hilfe integrieren	190
Neue Funktion verfassen	192
Funktion löschen	194
Funktion mit Schreibschutz versehen	195
Pipeline-Filter anlegen	196

Feststellen, ob ein Fehler aufgetreten ist	197
Fehler in Skripts und Funktionen abfangen	200
Eigene Fehler auslösen	203
Zusammenfassung	204

Genügt eine einzelne kurze PowerShell-Codezeile nicht mehr, um eine bestimmte Aufgabe durchzuführen, können Skripts eine Lösung sein. Skripts sind Textdateien mit der Erweiterung *.ps1*, die beliebig viele Zeilen PowerShell-Code enthalten dürfen, der beim Aufruf von oben nach unten durchlaufen und ausgeführt wird. Sie funktionieren also ganz ähnlich wie Batchdateien und sind immer dann empfehlenswert, wenn eine Aufgabe viele Zeilen Code erfordert und/oder regelmäßig immer wieder ausgeführt werden soll.

Mit Funktionen kann man ebenfalls mehrere Zeilen PowerShell-Code kapseln und sozusagen neue eigene »Cmdlets« hinzuerfinden. Funktionen ähneln Makros und fassen mehrere Zeilen Code zu einem neuen Befehl zusammen. Funktionen werden mitunter auch »Skript-Cmdlets« genannt, weil sie sich für den Anwender kaum von einem Cmdlet unterscheiden.

Funktionen liefern nicht nur innerhalb von Skripts neue Funktionalität. Starten Sie ein Skript »dot-sourced«, also mit einem vorangestellten Punkt, stehen anschließend alle darin definierten Funktionen zur Verfügung. Definiert man Funktionen in einem der speziellen Profilskripts, stehen diese sogar automatisch zur Verfügung, weil Profilskripts bei jedem Start der PowerShell automatisch ausgeführt werden.

Skripts verfassen

Sie möchten ein PowerShell-Skript verfassen, zum Beispiel, weil Sie eine komplexe Aufgabe lösen wollen und dazu mehr als nur ein oder zwei PowerShell-Befehlszeilen benötigen.

Lösung

PowerShell-Skripts sind reine Textdateien mit der Dateierweiterung *.ps1*. Im einfachsten Fall erstellen Sie ein neues Skript mit dem Windows-Editor:

```
PS > notepad meinskript.ps1
```

Verfassen Sie dann den Skriptcode und speichern Sie das Skript. Skripts dürfen dieselben Befehle enthalten, die Sie auch interaktiv in der Konsole einsetzen.

Mehr Komfort bietet das *Integrated Scripting Environment* (ISE), das Teil von PowerShell ist. Aus der PowerShell-Konsole starten Sie diesen Skripteditor mit dem Alias *ise*:

```
PS > ise
```

Sie erreichen *Windows PowerShell ISE* in Windows 7 bzw. Windows Server 2008 R2 auch über einen Rechtsklick auf das PowerShell-Symbol in der Taskleiste. Dieses Symbol ist nur zu sehen, wenn Sie entweder PowerShell gestartet oder das Symbol per Rechtsklick in der Taskleiste angeheftet haben.

Windows PowerShell ISE ist auf Windows-Servern nicht standardmäßig installiert und muss gegebenenfalls noch als optionales Windows-Feature in der Systemsteuerung aktiviert werden.

Hintergrund

Da PowerShell-Skripts reguläre Textdateien sind, dürfen Sie jeden beliebigen Texteditor zum Erstellen von PowerShell-Skripts einsetzen. Allerdings bietet der Windows-Editor keinerlei spezielle PowerShell-Unterstützung. Mehr Komfort liefern spezialisierte PowerShell-Editoren wie *Windows PowerShell ISE* und Editoren (teils kommerziell) von Drittanbietern wie *PowerShell Plus* von Idera oder *PowerGUI* von Quest.

Innerhalb von Skripten sind genau dieselben Befehle erlaubt wie in der interaktiven Konsole. Allerdings ist es in einem Editor wesentlich einfacher, Text zu editieren. Anders als in der interaktiven Konsole, wo immer nur eine Zeile zur Verfügung steht, dürfen Sie in Skripten den Code auf mehrere Zeilen verteilen. Das macht den Code übersichtlicher.

Nach einigen Zeichen wie dem Pipeline-Symbol (»|«) oder dem Aufzählungszeichen (»«,«) dürfen Sie die Zeile immer umbrechen:




```
Get-ChildItem $env:windir |  
Where-Object { $_.Length -gt 1MB }  
  
$liste = 1,8,12,19,  
22,26,77
```

Nach solchen Umbrüchen sind Kommentare erlaubt:

```
# alle Dateien aus dem Windows-Ordner auflisten...  
Get-ChildItem $env:windir |  
# ...und nur Dateien anzeigen, die größer sind als 1MB:  
Where-Object { $_.Length -gt 1MB }
```

Möchten Sie Zeilen an anderen Stellen umbrechen, setzen Sie das Backtick-Zeichen ein (»`«). Nach diesem Zeichen darf nur noch der Umbruch erfolgen. Kommentare zwischen den einzelnen Zeilen sind nicht erlaubt und auch hinter dem Backtick-Zeichen dürfen keine weiteren Zeichen folgen.

```
Get-ChildItem $env:windir -ErrorAction SilentlyContinue `~  
-Recurse -Filter *.log
```

Weil das Backtick-Zeichen über die Tastatur nur schwer zu erreichen ist (halten Sie die -Taste fest, drücken Sie das Zeichen rechts von  und dann ) und im Listing leicht übersehen wird, sollten Sie solche Umbrüche möglichst vermeiden.

Skripts starten

Sie wollen ein PowerShell-Skript ausführen.

Lösung

Geben Sie innerhalb der PowerShell-Umgebung den relativen oder absoluten Pfadnamen des Skripts ein:

```
PS > .\meinskript.ps1           # relativer Pfadname, Skript →
    liegt im aktuellen Ordner
PS > c:\skripts\meinskript.ps1  # absoluter Pfadname
```

Enthält der Pfad Leerzeichen oder Variablen, stellen Sie ihn in Anführungszeichen. Weil der Pfad nun ein Text ist, müssen Sie davor ein »&« und ein Leerzeichen stellen:

```
PS > & 'c:\meine skripts\mein skript.ps1'  # Pfad enthält Leerzeichen
PS > & "$home\mein skript.ps1"             # Pfad enthält Variablen, →
    doppelte Anführungszeichen nötig
PS > & $home\meinskript.ps1                 # Pfad enthält keine Leerzeichen, →
    keine Anführungszeichen nötig
```

WICHTIG

Ersetzen Sie das Zeichen »&« durch ».«, wenn der Gültigkeitsbereich des Skripts mit dem Gültigkeitsbereich der Konsole verschmelzen soll. Alle Variablen und Funktionen des Skripts stehen dann anschließend weiterhin in der Konsole zur Verfügung. Nötig ist das, wenn Sie ein Skript debuggen wollen oder wenn das Skript Funktionen definiert, die anschließend weiter in der Konsole verwendet werden sollen.

Falls PowerShell meldet, das Skript könne nicht ausgeführt werden, ändern Sie die Sicherheitseinstellung Ihrer PowerShell-Umgebung. Dies braucht nur einmal zu geschehen:

```
PS > Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned -Force
```

Möchten Sie ein Skript außerhalb der PowerShell-Umgebung starten, zum Beispiel als Verknüpfung oder als geplante Aufgabe, starten Sie zuerst *powershell.exe* und übergeben diesem Programm die folgenden Parameter:

```
powershell.exe -noprofile -nologo -executionpolicy bypass -file "c:\skripts\meinskript.ps1"
```

Hintergrund

Bevor PowerShell-Skripts überhaupt ausgeführt werden können, muss die Sicherheitseinstellung dies erlauben. *Get-ExecutionPolicy* meldet die aktive Sicherheitseinstellung und der folgende Befehl listet alle Sicherheitseinstellungen auf:

```
PS > Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
-----	-----
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	RemoteSigned
LocalMachine	Restricted

PowerShell wertet die Sicherheitseinstellungen von oben nach unten aus. Einstellungen, die auf *Undefined* eingestellt sind, werden ignoriert. Die obersten Einstellungen *MachinePolicy* und *UserPolicy* lassen sich nur zentral über Gruppenrichtlinien ändern. *Process* legt die Einstellung nur für die aktuelle Sitzung fest. *CurrentUser* betrifft nur den aktuellen Benutzer und wird in der Registrierungsdatenbank in *HKEY_CURRENT_USER* eingetragen. *Machine* gilt für alle Benutzer des Computers und wird in der Registrierungsdatenbank in *HKEY_LOCAL_MACHINE* gespeichert. Um diese Einstellung zu ändern, benötigen Sie lokale Administratorberechtigungen.

Set-ExecutionPolicy ändert die Sicherheitseinstellung. Mit dem Parameter *-Scope* legen Sie fest, auf welcher Ebene die Einstellung geändert werden soll. Der Parameter *-ExecutionPolicy* legt die Sicherheitseinstellung selbst fest. Erlaubt sind die Werte aus Tabelle 6.1.

Diese Zeile stellt die Sicherheitseinstellung für die aktuelle Sitzung um auf *Bypass*. Sie gilt nur in der aktuellen PowerShell-Sitzung:

```
PS > Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass -Force
```

Einstellung	Beschreibung
<i>Restricted</i>	Skripts dürfen nicht ausgeführt werden
<i>Default</i>	Standardeinstellung des Systems, entspricht <i>Restricted</i>
<i>AllSigned</i>	Alle Skripts müssen über eine gültige digitale Signatur verfügen. Unsignierte Skripts und Skripts mit ungültiger Signatur werden nicht ausgeführt.
<i>RemoteSigned</i>	Wie <i>AllSigned</i> , gilt aber nur für Skripts aus »öffentlichen Orten«. Dazu zählt PowerShell Skripts, die aus dem Internet heruntergeladen wurden, die aus E-Mail-Anhängen oder von Dateiservern stammen, die nicht zur eigenen Netzwerkdomäne gehören. Alle anderen Skripts dürfen auch ohne gültige Signatur ausgeführt werden.
<i>Unrestricted</i>	Alle Skripts dürfen ausgeführt werden. Stammt das Skript aus öffentlichen Orten, erscheint aber zunächst eine Sicherheitsabfrage.

Tabelle 6.1 Sicherheitseinstellungen für PowerShell-Skripts

Einstellung	Beschreibung
<i>Bypass</i>	Alle Skripts dürfen ausgeführt werden. PowerShell umgeht die Überprüfung der Skriptherkunft und Signatur.

Tabelle 6.1 Sicherheitseinstellungen für PowerShell-Skripts (*Fortsetzung*)

TIPP

Wollen Sie Skripts ohne Angabe eines relativen oder absoluten Pfadnamens aufrufen, müssen die Skripts in einem Ordner lagern, der in der Umgebungsvariable *Path* aufgeführt ist. So gehen Sie vor:

```
PS > md c:\meineskripts
PS > $env:path += ';c:\meineskripts'
```

Legen Sie nun im Ordner *c:\meineskripts* ein PowerShell-Skript an, dürfen Sie es unter Angabe seines Namens starten. Sie müssen nun keinen Pfad mehr angeben und können beim Aufruf auch auf die Dateierweiterung *.ps1* verzichten. Weil die Erweiterung der Umgebungsvariable *Path* nur für die aktuelle Sitzung gilt, sollten Sie sie in Ihr Profilskript aufnehmen (siehe unten).

Möchten Sie ein Skript von außerhalb der PowerShell-Umgebung starten, zum Beispiel als Verknüpfung auf dem Desktop oder als geplanter Task, rufen Sie *powershell.exe* auf und geben dahinter den Pfadnamen des Skripts an. Damit beim Start von PowerShell nicht Ihre automatischen Profilskripts geladen werden, verwenden Sie den Schalter *-noProfile*.

Normalerweise schließt das PowerShell-Fenster automatisch, sobald das Skript seine Arbeit erledigt hat. Möchten Sie das Fenster nicht schließen, geben Sie den Schalter *-noExit* an. Die PowerShell-Konsole bleibt dann nach Ausführung des Skripts geöffnet. Allerdings hat der Anwender nun auch die Möglichkeit, weitere PowerShell-Befehle einzugeben.

Eine Übersicht sämtlicher Startoptionen erhalten Sie, indem Sie in der PowerShell-Konsole die Hilfe zu *powershell.exe* abrufen:

```
PS > powershell.exe /?
```

Skripts automatisch ausführen

Sie möchten beim Start von PowerShell automatisch Skripts ausführen, um in diesen Skripts Ihre Arbeitsumgebung zu gestalten und zum Beispiel die Konsole anzupassen oder PowerShell-Snap-Ins zu laden.

Lösung

Legen Sie ein Skript an einem der vier Orte ab, die PowerShell bei jedem Start überprüft. Sind dort Skripts vorhanden, führt PowerShell sie bei jedem Start automatisch aus:


```
PS > $profile
C:\Users\Benutzername\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
PS > $profile.CurrentUserCurrentHost
C:\Users\Benutzername\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
PS > $profile.CurrentUserAllHosts
C:\Users\Benutzername\Documents\WindowsPowerShell\profile.ps1
PS > $profile.AllUsersCurrentHosts
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
PS > $profile.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Möchten Sie Ihr persönliches Profilskript automatisch anlegen, gehen Sie so vor:

```
PS > md ($Split-Path $profile) -ea 0 | Out-Null; notepad $profile
```

Hintergrund

Profilskripts sind normale Skripts, die aber von PowerShell bei jedem Start automatisch »dot-sourced«¹ ausgeführt werden. Alle Variablen und Funktionen in diesen Skripts stehen also anschließend in Ihrer PowerShell-Umgebung zur Verfügung, ebenso wie alle von dort aus nachgeladenen Module oder Snapins. Mit den Profilskripts lassen sich also Einstellungen festlegen, die automatisch in Ihren PowerShell-Sitzungen wirksam sein sollen.

Profil	Beschreibung
<i>\$profile.AllUsersAllHosts</i>	Gemeinsames Profil für alle Benutzer
<i>\$profile.AllUsersCurrentHosts</i>	Gemeinsames Profil für alle Benutzer, gültig nur für den aktuellen Host, also beispielsweise <i>powershell.exe</i>
<i>\$profile.CurrentUserAllHosts</i>	Profil des aktuellen Benutzers
<i>\$profile.CurrentUserCurrentHost</i> <i>\$profile</i>	Profil des aktuellen Benutzers, gültig nur für den aktuellen Host, also beispielsweise <i>powershell.exe</i>

Tabelle 6.2 Die PowerShell-Profile

Skripts in der Pipeline verwenden

Sie möchten ein Skript innerhalb der PowerShell-Pipeline verwenden. Das Skript soll die Elemente der Pipeline in Echtzeit bearbeiten.

Lösung

Verwenden Sie im Skript einen *process*-Block. Dieser Skriptblock wird für jedes Element der Pipeline in Echtzeit aufgerufen. Das jeweilige Pipeline-Element findet sich in der Variablen *\$_*. In einem optionalen *begin*-Block stehen Anweisungen, die einmalig am Anfang des Skripts aus-

geführt werden sollen. In einem optionalen *end*-Block stehen Anweisungen, die einmalig zum Schluß ausgeführt werden sollen. Anweisungen außerhalb dieser drei Blöcke sind nicht erlaubt.

Das folgende Skript bearbeitet die Pipeline-Ergebnisse von *dir* und färbt Dateien mit der Erweiterung *.exe* im Ordnerlisting rot.

```
begin
{
  $farbe = $host.UI.RawUI.ForegroundColor
}

process {
  if ($_.Extension -eq '.exe')
  {
    $host.UI.RawUI.ForegroundColor = 'Red'
  } else {
    $host.UI.RawUI.ForegroundColor = $farbe
  }
  $_
}

end {
  $host.UI.RawUI.ForegroundColor = $farbe
}
```

So rufen Sie das Skript innerhalb der Pipeline auf (in diesem Beispiel wird davon ausgegangen, dass das Skript als *c:\markexe.ps1* gespeichert wurde):

```
PS > dir $env:windir | c:\markexe.ps1
```

Hintergrund

Damit ein Skript pipelinefähig wird, verlagern Sie denjenigen Codeteil in einen *process*-Block, der für jedes empfangene Ergebnis des vorangehenden Pipeline-Befehls wiederholt werden soll. Innerhalb des *process*-Blocks wird das aktuell empfangene Objekt in der Variablen *\$_* zur Verfügung gestellt.

Sobald Sie einen *process*-Block in einem Skript verwenden, dürfen keine Codeteile mehr außerhalb dieses Blocks stehen. Soll Code vor Beginn der Pipelineschleife ausgeführt werden, verlagern Sie ihn in einen *begin*-Block. Code, der nach Abschluss der Pipelineoperation ausgeführt werden soll, wird in einen *end*-Block verlagert.

Setzen Sie ein Skript ohne *process*-Block innerhalb der Pipeline ein, empfängt es die Ergebnisse des Vorgängerbefehls in der automatischen Variable *\$input*. Es blockiert so allerdings die Pipeline, weil die Pipeline zuerst darauf wartet, bis der Vorgängerbefehl seine Arbeit vollständig erledigt hat. Erst danach wird sein Ergebnis in *\$input* an das Skript weitergereicht.

Enthält ein Skript die Skriptblöcke *begin*, *process* und *end*, arbeitet es dagegen im schnellen Streamingmodus und bearbeitet die Pipeline-Ergebnisse in Echtzeit.

Fügen Sie einen *param()*-Block am Beginn des Skripts ein, kann es die Ergebnisse eines Vorgängerbefehls noch flexibler verarbeiten. Das folgende Skript deklariert zwei Parameter, von denen einer die Ergebnisse des Vorgängerbefehls empfangen kann, aber nicht muss:

```
param(
    [Parameter(ValueFromPipeline=$true)]
    $betrag,
    $kurs
)

process {
    $betrag * $kurs
}
```

Auf diese Weise kann das Skript sowohl eigenständig als auch innerhalb der Pipeline sinnvoll eingesetzt werden. Setzen Sie es eigenständig ein, können Sie den Parameter *-betrag* selbst festlegen.

```
PS > C:\meinskript.ps1 -betrag 100 -kurs 2
200
```

Innerhalb der Pipeline eingesetzt, empfängt dasselbe Skript den Betrag hingegen vom vorangehenden Befehl oder den Daten, die Sie am Anfang der Pipeline hinterlegt haben. Im folgenden Aufruf rechnet das Skript also die Werte 1 bis 5 um:

```
PS > 1..5 | C:\meinskript.ps1 -kurs 3
3
6
9
12
15
```

Alternativ kann das Skript die Eingangsparameter auch aus Objekteigenschaften auslesen. In diesem Fall können mehrere Werte gleichzeitig aus den Ergebnissen des Vorgängerbefehls übergeben werden. Das folgende Skript empfängt die Parameter *Length* und *Name* gleichzeitig:

```
param(
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    $length,
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    $Name
)

process {
    "$name ist $length Bytes gross"
}
```

Empfängt das Skript nun Objekte, die sowohl eine Eigenschaft *Length* als auch eine Eigenschaft *Name* besitzen, werden die Inhalte dieser Eigenschaften automatisch an das Skript übergeben:

```
PS > Dir $env:windir | c:\meinskript.ps1
bfsvc.exe ist 71168 Bytes gross
bootstat.dat ist 67584 Bytes gross
DirectX.log ist 197 Bytes gross
DPINST.LOG ist 6518 Bytes gross
(...)
```

Empfängt das Skript dagegen Objekte, die diese Eigenschaften nicht besitzen, bleiben die Parameter leer. Deshalb ist der Parameter *–Length* leer, wenn das Skript Ordner empfängt, und nur gefüllt, wenn es Dateien empfängt.

Möchten Sie, dass ein Parameter Eingaben von mehreren verschiedenen Objekteigenschaften empfangen kann, definieren Sie für den Parameter weitere Aliasnamen. Würden Sie den *param()*-Block beispielsweise folgendermaßen formulieren, könnte das Skript auch *Process*-Objekte empfangen:

```
param(
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    [Alias('WorkingSet')]
    $length,
    [Parameter(ValueFromPipelineByPropertyName=$true)]
    $Name
)
```

Dem Parameter *–Length* ist jetzt der Alias *WorkingSet* zugewiesen. Deshalb würde das Skript bei Prozessen die Größe aus der Eigenschaft *WorkingSet* und nicht mehr aus der Eigenschaft *Length* lesen, denn die Eigenschaft *Length* gibt es bei Prozessen nicht:

```
PS > Get-Process | c:\meinskript.ps1
DataCardMonitor ist 1560576 Bytes gross
dwm ist 30646272 Bytes gross
explorer ist 17252352 Bytes gross
(...)
```

Rückgabewert eines Skripts festlegen

Sie möchten aus einem PowerShell-Skript heraus einen Rückgabewert festlegen, zum Beispiel, damit der Aufrufer des Skripts erkennen kann, ob das Skript fehlerfrei ausgeführt wurde.

Lösung

Beenden Sie das Skript explizit mit der Anweisung *Exit* und geben Sie darin den gewünschten Rückgabewert an. Als Rückgabewert kommt nur eine Ganzzahl infrage.

Das folgende Beispiel legt ein Miniskript an, das mit *Read-Host* nach einer Zahl fragt und diese dann als eigenen Rückgabewert verwendet.

```
$wert = [Int] (Read-Host "Geben Sie eine Zahl an")  
Exit($wert)
```

```
PS > c:\meinskript.ps1  
Geben Sie eine Zahl an: 7  
PS > $lastexitcode  
7
```

Hintergrund

Selbstverständlich kann ein Skript beliebige Informationen zurückliefern, indem Sie Texte oder Ergebnisse eines Befehls einfach »liegenlassen«. Diese werden dann in die PowerShell-Konsole ausgegeben.

Wird ein PowerShell-Skript nicht aus PowerShell heraus aufgerufen, sondern beispielsweise als *Geplante Aufgabe* oder von Batchdateien, kann der Aufrufer normale Ausgaben des Skripts nicht auswerten. Stattdessen benötigt der Aufrufer einen sogenannten »Error Level«.

Diesen kann das Skript mit der Anweisung *Exit* festlegen. Wie bei jeder konsolenbasierten Anwendung findet sich der Error Level des letzten Aufrufs in der PowerShell-Variable *\$LASTEXITCODE*, die man wie im Beispiel deshalb für Tests einsetzen kann.

Rufen Sie ein PowerShell-Skript von außerhalb auf, beispielsweise aus einer Batchdatei heraus, wird der Rückgabewert des Skripts in *%ERRORLEVEL%* gemeldet. Geben Sie die folgenden Zeilen in das Fenster einer Eingabeaufforderung ein, das Sie mit *cmd.exe* starten:

```
> Powershell.exe -nopprofile -executionpolicy Bypass -file c:\meinskript.ps1  
Geben Sie eine Zahl an: 88  
> Echo %errorlevel%  
88
```

Optionale Parameter definieren

Sie möchten, dass der Anwender einem Skript Parameter übergeben darf, aber nicht muss.

Lösung

Fügen Sie am Start des Skripts einen *param()*-Block ein und geben Sie darin kommasepariert alle Parameter an, die das Skript vom Aufrufer empfangen soll.

```
param(  
    $dollar,  
    $exchangerate = 0.75  
)  
  
$dollar * $exchangerate
```

Der Anwender kann dem Skript nun zwei Parameter übergeben. Weil dem Parameter – *ExchangeRate* ein Vorgabewert zugewiesen ist, braucht der Anwender diesen Parameter nicht unbedingt anzugeben. Tut er es nicht, enthält die Variable *\$exchangerate* im Skript den Vorgabewert. Gibt der Anwender den Parameter an, enthält *\$exchangerate* den vom Anwender festgelegten Wert:

```
PS > c:\meinskript.ps1 -dollar 100
75
PS > c:\meinskript.ps1 -dollar 100 -exchangerate 3
300
```

Hintergrund

Die im *param()*-Block aufgeführten Variablen werden automatisch zu Parameternamen, für die dieselben Regeln gelten wie für Cmdlet-Parameter. Sie dürfen die Parameternamen also abkürzen, solange sie eindeutig bleiben, und die Argumente auch ohne Parameternamen angeben, wenn Sie sich an die richtige Reihenfolge halten:

```
PS > c:\meinskript.ps1 100 2.5
250
PS > c:\meinskript.ps1 200 -exch 7
1400
PS > c:\meinskript.ps1 -e 2 -d 100
200
```

Achten Sie bei der Namensgebung der Parameter darauf, sprechende Parameternamen zu verwenden, die sich möglichst an üblichen Parameternamen orientieren. Wenn Sie beispielsweise einen Dateinamen erfragen, nennen Sie diesen Parameter –*Path* und nicht –*Name* oder etwa –*d*. Weil die Anwender die Parameternamen später per AutoVervollständigung abrufen können, sollten die Parameternamen intuitiv ihre Bedeutung erschließen lassen.

Belegen Sie möglichst viele Parameter mit sinnvollen Vorgabewerten, damit die Anwender nicht gezwungen sind, Parameter anzugeben, sondern alternativ die Vorgabewerte nutzen können.

Möchten Sie einen Vorgabewert mit einem dynamischen (jedes Mal frisch berechneten) Wert versehen, weisen Sie dem Parameter einen Vorgabewert zu, der das Ergebnis eines PowerShell-Befehls ist. Im folgenden *param()*-Block wird dem Parameter –*OutFile* beispielsweise ein automatisch generierter Dateiname mit einem aktuellen Zeitstempel im *Temp*-Ordner zugewiesen:

```
param(
    $OutFile = "$env:temp\logfile{0}.txt" -f (Get-Date -Format →
        'yyyyMMddHHmmss')
)

"Lege Datei hier an: $OutFile"
```

Rufen Sie dieses Skript ohne Parameter auf, wird ein aktueller temporärer Dateiname automatisch generiert:

```
PS > c:\meinskript.ps1
Lege Datei hier an: C:\Users\w7-pc9\AppData\Local\Temp\logfile20110203200254.txt
PS > c:\meinskript.ps1 -OutFile c:\ausgabe.txt
Lege Datei hier an: c:\ausgabe.txt
PS > c:\meinskript.ps1 test
Lege Datei hier an: C:\Users\w7-pc9\AppData\Local\Temp\logfile20110203200308.txt
```

Zwingend erforderliche Parameter definieren

Sie möchten dafür sorgen, dass ein Parameter zwingend vom Anwender angegeben werden muss.

Lösung

Deklariieren Sie den Parameter als zwingend erforderlich. Wird der Parameter später nicht vom Anwender angegeben, fragt PowerShell nach und gibt dem Anwender Gelegenheit, den Parameter nachzuliefern.

```
param(
    [Parameter(Mandatory=$true)]
    $dollar,
    $exchangerate = 0.75
)
$exchangerate * $dollar
```

```
PS > c:\meinskript.ps1
```

```
Cmdlet meinskript.ps1 an der Befehlspipelineposition 1
Geben Sie Werte für die folgenden Parameter an:
dollar: 200
150
```

Hintergrund

Um PowerShell Besonderheiten eines Parameters mitzuteilen, werden Parameter-Attribute verwendet. Sie sind stets nach dem Muster `[Parameter(Liste)]` aufgebaut und werden direkt vor dem Parameter angegeben. Weil Parameterattribute direkt zum Parameter gehören, wird zwischen Attribut und Parametername auch kein Komma gesetzt. Ein Zeilenumbruch ist aber erlaubt und aus Gründen der Übersichtlichkeit auch sinnvoll.

Deklariieren Sie einen Parameter als zwingend erforderlich, verhält sich PowerShell genauso wie bei Cmdlets. Der Anwender erhält also eine automatische Meldung und die Gelegenheit, den Parameterwert nachzuliefern, wenn er ihn nicht angegeben hat.

WICHTIG

Parameterinhalte, die von PowerShell automatisch nachträglich abgefragt worden sind, entsprechen immer dem Datentyp *String* (Text). Erwartet der Parameter aber eigentlich keinen Text, sondern Zahlen, sollte der Parameter unbedingt streng typisiert werden – also den erwarteten Datentyp in eckigen Klammern ausdrücklich angeben. Nur dann konvertiert PowerShell nachträglich erfragte Parameterinhalte automatisch in den gewünschten Datentyp.

```
param(  
    [Parameter(Mandatory=$true)]  
    [Double]  
    $dollar,  
    $exchangerate = 0.75  
)
```

Switch-Parameter definieren

Sie möchten Ihre Funktion mit einem *Switch*-Parameter ausstatten (*Ja/Nein*-Schalter).

Lösung

Switch-Parameter erhalten Sie, indem Sie den Typ des Parameters auf *[Switch]* einstellen. Wird der *Switch*-Parameter vom Anwender angegeben, enthält der Parameter den Wert *\$true*, sonst *\$false*. Mit einer Bedingung kann dann geprüft werden, ob der *Switch*-Parameter angegeben wurde:

```
param(  
    [Switch]  
    $ShowAll  
)  
  
If ($ShowAll) {  
    'ShowAll was specified'  
} else {  
    'Default behavior'  
}
```

```
PS > c:\meinskript.ps1
```

```
Default behavior
```

```
PS > c:\meinskript.ps1 -ShowAll
```

```
ShowAll was specified
```


Hintergrund

Switch-Parameter funktionieren wie Ja/Nein-Schalter und sind, anders als andere Parameter-typen, keine Schlüssel-Wert-Paare. Stattdessen kann der Anwender den *Switch*-Parameter angeben oder nicht.

Wird der Parameter angegeben, kann die Funktion darauf reagieren. Häufig werden *Switch*-Parameter in Form eines *–Force*-Switches dazu verwendet, um zusätzliche oder spezielle Funktionalitäten bereitzustellen. Geben Sie beispielsweise bei *Get-ChildItem* (Alias: *Dir*) den *Switch*-Parameter *–Force* an, zeigt das Cmdlet auch versteckte Dateien an.

Eine Variable vom Typ *Switch* ist in Wirklichkeit ein Objekt mit der Eigenschaft *isPresent*. Wurde der *Switch*-Parameter beim Funktionsaufruf vom Anwender angegeben, liefert die Eigenschaft *isPresent* den Wert *\$true* zurück, sonst *\$false*.

Der einfachste Weg, einen *Switch*-Parameter zu überprüfen, ist eine *If*-Bedingung, die den *Switch*-Parameter auswertet.

```
If ($ShowAll) {  
    'ShowAll was specified'  
}
```

Es ist nicht notwendig, explizit seine Eigenschaft *IsPresent* anzugeben, und es muss auch kein Vergleich durchgeführt werden, weil das Ergebnis bereits ein boolescher Wert ist. Rein formal betrachtet sieht die Bedingung folgendermaßen aus:

```
If ($ShowAll.IsPresent -eq $true) {  
    'ShowAll was specified'  
}
```

ACHTUNG

Die Namen von *Switch*-Parametern dürfen nicht mit einer Zahl beginnen.

Die folgende Funktion *Get-ProcessList* macht sich *Switch*-Parameter zunutze, um Prozesse einer 64-Bit-Maschine wahlweise als Liste von 32-Bit- oder echten 64-Bit-Prozessen auszugeben:

```
function Get-ProcessList {  
    param(  
        [switch]  
        $Include32Bit,  
        [switch]  
        $Include64Bit  
    )  
  
    if ($Include32Bit) {  
        Get-Process |  
        Where-Object {  
            ($_  
                Select-Object -ExpandProperty Modules -ea 0 |  
                Select-Object -ExpandProperty ModuleName) -contains 'wow64.dll'  
            }  
        }  
    }  
}
```

```

    }
  }
  if ($Include64Bit) {
    Get-Process |
    Where-Object {
      ($_ |
        Select-Object -ExpandProperty Modules -ea 0 |
        Select-Object -ExpandProperty ModuleName) -notcontains 'wow64.dll'
      }
    }
  }
}

```

PS > **Get-ProcessList -Include32Bit**

Parameter validieren

Sie möchten sicherstellen, dass der Anwender einem Parameter nur bestimmte Informationen übergeben darf.

Lösung

Wollen Sie einen Parameter auf einen bestimmten Datentyp festlegen, stellen Sie den Datentyp in eckigen Klammern vor den Parameter. Der folgende *param()*-Block definiert einen Parameter *-datum*, dem ein gültiges Datum übergeben werden muss:

```

param(
    [DateTime]
    $datum
)

```

Andernfalls setzen Sie Validierungsattribute ein. Der Parameter *-Name* darf im folgenden Beispiel nicht kürzer als 3 und nicht länger als 10 Zeichen sein:

```

Param(
    [ValidateLength(3,10)]
    $name
)

```

Sollen Zahlenbereiche überprüft werden, setzt man *ValidateRange* ein. Das folgende Attribut erlaubt nur Zahlen zwischen 10 und 30:

```
[ValidateRange(10,30)]
```

Dürfen nur bestimmte Schlüsselbegriffe angegeben werden, setzen Sie *ValidateSet* ein. Das folgende Attribut erlaubt nur die Eingaben *Error*, *Warning* oder *Information*:

```
[ValidateRange('Error','Warning','Information')]
```

Für die Validierung dürfen auch reguläre Ausdrücke eingesetzt werden. Das Validierungsattribut *ValidatePattern* erlaubt die Eingabe von *PC1* bis *PC499*:

```
[ValidatePattern('^PC[0-4]{0,1}[0-9]{0,2}$')]
```

Am flexibelsten sind *ValidateScript*-Attribute. Hier kann ein eigenes kleines Skript konzipiert werden, das den Wert überprüft. Im Folgenden wären nur Eingaben erlaubt, die einer existierenden Datei im Windows-Ordner entsprechen:

```
[ValidateScript({(Dir $env:windir | Select-Object -Expand Name) →  
-contains $_})]
```

Hintergrund

Weisen Sie einem Parameter einen Datentyp zu, versucht PowerShell automatisch, die Benutzereingabe in diesen Datentyp umzuwandeln. Ist die Eingabe nicht umwandelbar, meldet PowerShell einen aussagekräftigen Fehler.

HINWEIS Hierbei kann es zu unerwünschten Nebenwirkungen kommen. Gibt der Anwender beispielsweise ein deutsches Datum als Text ein und ist der Parameter auf den Datentyp *DateTime* festgelegt, wandelt PowerShell das Datum zwar möglicherweise um, setzt dabei aber das »kulturneutrale« US-amerikanische Datumsformat ein. Monat und Tag sind anschließend vertauscht. In diesem Fall ist eine Skriptvalidierung besser geeignet. Das folgende Attribut akzeptiert nur deutsche Datumsformate (beziehungsweise Formate, die Ihren Systemeinstellungen entsprechen):

```
[ValidateScript({$_ -as [DateTime]})]
```

Achten Sie darauf: Die Validierung erfolgt erst, nachdem der Parameter umgewandelt ist. Wenn Sie den Parameter also außerdem auf den Datentyp *DateTime* festlegen, wird die Eingabe zuerst mit dem amerikanischen Format umgewandelt und ist danach ein *DateTime*-Wert. Das Validierungsskript ergibt jetzt immer *\$true* und ist also sinnlos geworden.

Mehrere Werte pro Parameter übergeben

Sie möchten einem Parameter mehrere Werte zuweisen und diese Werte anschließend der Reihe nach bearbeiten.

Lösung

Weisen Sie dem Parameter die Werte als kommaseparierte Liste zu und leiten Sie den Parameter innerhalb des Skripts an *ForEach-Object* weiter. Das folgende Skript liest die letzten zehn Error-Ereignisse aus allen Ereignislogbüchern, die dem Parameter *-LogName* übergeben werden:

```
param(
    $logname,
    $anzahl = 10
)

$logname |
ForEach-Object {
    $log = $_
    Get-EventLog -LogName $_ -EntryType Error -Newest $anzahl |
    ForEach-Object {
        $_ | Add-Member -MemberType NoteProperty -Name LogName -Value $log →
        -PassThru |
        Select-Object LogName, EventID, Message, Source
    }
} |
Sort-Object TimeGenerated -Descending |
Select-Object -First $anzahl
```

Rufen Sie das Skript mit mehreren Logbuchnamen auf, werden die letzten zehn Fehler aus allen angegebenen Logbüchern abgerufen:

```
PS > C:\meinskript.ps1 -logname system, application
```

Hintergrund

Parametern dürfen mehrere Werte in Form von Arrays (Feldern) übergeben werden. Arrays werden mit dem Zeichen »« angelegt. Weisen Sie einem Parameter also eine kommaseparierte Liste zu, empfängt der Parameter ein Array mit den übergebenen Werten.

Damit das Skript die einzelnen übergebenen Werte einzeln bearbeitet, werden diese an *ForEach-Object* weitergeleitet. Der Skriptblock von *ForEach-Object* empfängt nun nacheinander die einzelnen Werte und der Skriptblock wird für jeden Wert erneut aufgerufen. In der Variable *\$_* steht dabei der jeweils zu bearbeitende Wert.

Über Validierungsattribute wie *ValidateCount* lässt sich die Anzahl der Werte beschränken, die einem Parameter zugewiesen werden darf. Das folgende Skript akzeptiert zwar mehrere Computernamen, aber nicht mehr als fünf:

```
param(  
    [ValidateCount(1,5)]  
    $computername  
)  
  
$computername |  
ForEach-Object {  
    "bearbeite $_"  
}
```

Bei mehr als fünf Computernamen wird ein Fehler ausgegeben:

```
PS > c:\meinskript.ps1 -computername pc1,pc2,pc3,pc4,pc5  
bearbeite pc1  
bearbeite pc2  
bearbeite pc3  
bearbeite pc4  
bearbeite pc5  
PS > c:\meinskript.ps1 -computername pc1,pc2,pc3,pc4,pc5,pc6  
Meinskript.ps1 : Das Argument für den Parameter "computername" kann nicht überprüft werden.  
Die Anzahl der angegebenen Argumente (6) überschreitet die maximal zulässige Anzahl von  
Argumenten (5). Geben Sie weniger als 5 Argumente an, und führen Sie dann den Befehl erneut  
aus.
```

ACHTUNG Wenn Sie einen Parameter streng typisieren, ihm also einen Datentyp zuweisen, kann die Umwandlung verhindern, dass mehrere Argumente zugewiesen werden können. Ordnen Sie im Beispiel oben dem Parameter `$computername` den Datentyp `String` zu:

```
param(  
    [ValidateCount(1,5)]  
    [String]  
    $computername  
)
```

Das Verhalten des Skripts ändert sich nun schlagartig:

```
PS > c:\meinskript.ps1 -computername pc1,pc2,pc3,pc4,pc5  
Meinskript.ps1 : Das Argument für den Parameter "computername" kann nicht überprüft werden.  
Die Anzahl der angegebenen Argumente (19) überschreitet die maximal zulässige Anzahl von  
Argumenten (5). Geben Sie weniger als 5 Argumente an, und führen Sie dann den Befehl erneut  
aus.
```

Tatsächlich wurde nun die gesamte Eingabe als ein einzelner Text interpretiert und seine Länge (19 Zeichen) ist länger als die erlaubte Länge von 5. PowerShell interpretiert die Eingabe nun als Array mit einzelnen Zeichen. Der richtige Datentyp in diesem Fall hätte `[String[]]` und nicht `[String]` gelautet, also ein Array von Strings und nicht ein einzelner String.

Hilfe integrieren

Sie möchten, dass ein Skript dem Anwender Hilfeinformationen liefert, wenn dieser mit *Get-Help* Hilfestellung anfordert.

Lösung

Fügen Sie am Anfang des Skripts einen Kommentarblock ein, der dem folgenden Muster entspricht:

```
<#  
.SYNOPSIS  
    Eine kurze Beschreibung.  
.DESCRIPTION  
    Eine ausführliche Beschreibung  
.PARAMETER betrag  
    Der Betrag der umgerechnet werden soll. Dieser Parameter ist  
    zwingend und muss angegeben werden.  
.PARAMETER wechsellkurs  
    Der Wechselkurs für die Umrechnung. Dieser Parameter ist  
    optional. Geben Sie diesen Parameter nicht an, wird der  
    Standardwert 0.98 verwendet.  
.EXAMPLE  
    .\skript.ps1 -betrag 100  
    Rechnet den Betrag mit dem Standardwechselkurs um  
.EXAMPLE  
    .\skript.ps1 -betrag 100 -wechselkurs 1.2  
    Rechnet den Betrag mit dem angegebenen Wechselkurs um  
.EXAMPLE  
    .\skript.ps1 100 1.2  
    Rechnet den Betrag mit dem angegebenen Wechselkurs um und  
    gibt die Argumente positional an.  
.NOTES  
    Weitere Hinweise  
.LINK  
    http://www.powershell.de  
#>
```

Hintergrund

Sofern Sie am Beginn eines Skripts einen Kommentarblock einfügen, der genauso formatiert ist wie oben angegeben, macht PowerShell diese Informationen über *Get-Help* abrufbar. Der Anwender kann dann also dieselbe komfortable Hilfestellung verwenden wie von den mitgelieferten Cmdlets gewohnt.

```
PS > Get-Help C:\meinskript.ps1
```

NAME

C:\meinskript.ps1

ÜBERSICHT

Eine kurze Beschreibung.

SYNTAX

C:\meinskript.ps1 [<CommonParameters>]

BESCHREIBUNG

Eine ausführliche Beschreibung

VERWANDTE LINKS

<http://www.powershell.de>

HINWEISE

Zum Aufrufen der Beispiele geben Sie Folgendes ein:

"get-help C:\meinskript.ps1 -examples".

Weitere Informationen erhalten Sie mit folgendem Befehl:

"get-help C:\meinskript.ps1 -detailed".

Technische Informationen erhalten Sie mit folgendem Befehl:

"get-help C:\meinskript.ps1 -full".

Mit dem Parameter *-examples* stehen die hinterlegten Beispiele zur Verfügung:

```
PS > Get-Help C:\meinskript.ps1 -examples
```

NAME

C:\meinskript.ps1

ÜBERSICHT

Eine kurze Beschreibung.

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

C:\PS>.\skript.ps1 -betrag 100

Rechnet den Betrag mit dem Standardwechsellkurs um

----- BEISPIEL 2 -----

C:\PS>.\skript.ps1 -betrag 100 -wechselkurs 1.2

Rechnet den Betrag mit dem angegebenen Wechselkurs um

----- BEISPIEL 3 -----

```
C:\PS>.\skript.ps1 100 1.2
```

Rechnet den Betrag mit dem angegebenen Wechselkurs um und gibt die Argumente positional an.

Neue Funktion verfassen

Sie möchten eine neue PowerShell-Funktion erstellen, zum Beispiel, weil Sie aus den Grundbestandteilen der PowerShell-Sprache eigene neue Befehle erstellen möchten.

Lösung

Verwenden Sie die *Function*-Konstruktion:

```
Function MeinName {  
  
}
```

Ersetzen Sie *MeinName* durch den Namen für Ihre Funktion. Obwohl keine Pflicht, sollte sich der Name einer Funktion denselben Regeln unterwerfen wie Cmdlets. Der Name sollte also aus einem Verb, einem Bindestrich und einem Nomen bestehen (»Tätigkeit-Tätigkeitsbereich«), und das Verb sollte einem der zugelassenen Verben entsprechen, die Sie mit *Get-Verb* abrufen können.

Innerhalb des Skriptblocks der Funktion haben Sie dieselben Möglichkeiten wie bei einem Skript, können also mit einem *param()*-Block Funktionsparameter definieren und eine Hilfefunktion einrichten.

Hintergrund

Funktionen sind im Grunde benannte Skriptblöcke: Sie verknüpfen also einen Namen mit einem ausführbaren Skriptblock in geschweiften Klammern, und wenn Sie später den Namen eingeben, wird der zugehörige Skriptblock ausgeführt:

```
PS > Function meineFunktion { 'Hallo' }  
PS > meineFunktion  
Hallo
```

Der Skriptblock selbst besteht aus geschweiften Klammern, die PowerShell-Anweisungen enthalten. Hinter den Funktionen stecken technisch gesehen »unbenannte Skriptblöcke«, die Sie zum Beispiel in einer Variablen aufbewahren und mit der *&*-Anweisung starten:


```
PS > $a = { 'Hallo' }  
PS > & $a  
Hallo
```

Was Sie genau im Skriptblock tun, bleibt Ihnen überlassen. Ein Skriptblock kann alles enthalten, was auch ein Skript enthalten darf, also beispielsweise einen *param()*-Block, um Parameter für die Funktion festzulegen, und einen Hilfeblock, damit die Funktion dem Anwender Hilfestellung bieten kann.

Die folgende Funktion generiert einen Excel-Report mit den Fehler-Ereignislogbucheinträgen der letzten 24 Stunden. Die Eckdaten lassen sich über Parameter frei wählen:

```
Function Get-EventReport {  
    param(  
        $Logname='System',  
        $EntryType='Error',  
        $Stunden=24,  
        $OutFile="$env:temp\log{0}.csv" -f (Get-Date -format →  
        'ddMMyyyyHHmmss')  
    )  
  
    $stichtag = (Get-Date) - (New-Timespan -hours $Stunden)  
  
    Get-EventLog -LogName $Logname -EntryType $EntryType -After $stichtag |  
    Export-CSV -Path $OutFile -NoTypeInfo -UseCulture -Encoding UTF8  
  
    Invoke-Item $OutFile  
}
```

Der folgende Aufruf würde einen Excel-Report mit den Warnungen im Systemlogbuch der letzten 72 Stunden erstellen:

```
PS > Get-Eventreport -EntryType Warning -Stunden 72
```

Die Lebensdauer einer Funktion richtet sich danach, wo Sie die Funktion definieren. Definieren Sie die Funktion in der PowerShell-Konsole, steht sie so lange zur Verfügung, bis Sie entweder die Konsole schließen oder die Funktion löschen oder überschreiben. Wird die Funktion innerhalb eines PowerShell-Skripts definiert, steht die Funktion nur so lange zur Verfügung, wie das Skript läuft, es sei denn, Sie starten das Skript »dot-sourced« (mit einem führenden Punkt und einem Leerzeichen vor dem Pfadnamen des Skripts) oder setzen vor den Funktionsnamen das Schlüsselwort »global:«.

Möchten Sie nützliche Funktionen permanent in allen PowerShell-Konsolen verfügbar machen, definieren Sie sie in einem der Profilskripts. Diese Profilskripts werden beim Start der PowerShell automatisch geladen und dot-sourced ausgeführt, um Ihre persönliche Arbeitsumgebung festzulegen. Alle darin definierten Funktionen stehen also künftig automatisch in jeder PowerShell-Konsole zur Verfügung.

PowerShell unterscheidet nicht zwischen Prozeduren (liefern keine Rückgabewerte) und Funktionen (liefern Rückgabewerte). Eine PowerShell-Funktion kann Ergebnisse zurückliefern, muss das aber nicht tun. Die Rückgabewerte einer Funktion bestehen aus denjenigen Objekten, die die Funktion »liegen lässt«. Es können beliebig viele sein. Bei mehr als einem Rückgabewert verpackt PowerShell die Ergebnisse automatisch als Feld (Array):

```
PS > Function FunktionMitErgebnis {  
>> 'Rückgabewert 1'  
>> 123  
>> Get-Date  
>> }  
>>
```

In diesem Beispiel gibt die Funktion drei Ergebnisse zurück.

```
PS > FunktionMitErgebnis  
Rückgabewert 1  
123  
  
Montag, 14. Februar 2011 13:01:00
```

Wenn eine Funktion mehr als ein Ergebnis liefert, verpackt PowerShell das Ergebnis in ein Feld (Array). Über dessen *Count*-Eigenschaft fragen Sie ab, wie viele Ergebnisse im Feld lagern. Über die eckigen Klammern greifen Sie mit einem Index auf einzelne Ergebnisse zu. Das erste Ergebnis trägt den Index 0. Negative Indizes zählen rückwärts, sodass der Index -1 das letzte Ergebnis liefert:

```
PS > $ergebnis = FunktionMitErgebnis  
PS > $ergebnis.Count  
3  
PS > $ergebnis[0]  
Rückgabewert 1  
PS > $ergebnis[-1]  
  
Montag, 14. Februar 2011 13:01:00
```

Funktion löschen

Sie möchten eine zuvor definierte Funktion löschen.

Lösung

Verwenden Sie *Remove-Item* und löschen Sie die Funktion im virtuellen Laufwerk *Function*:

```
PS > Function Test { 'Hallo' }
PS > Test
Hallo
PS > Remove-Item Function:test
PS > Test
Die Benennung "Test" wurde nicht als Cmdlet, Funktion, ausführbares Programm oder
Skriptdatei erkannt. Überprüfen Sie die Benennung, und versuchen Sie es erneut.
Bei Zeile:1 Zeichen:4
+ Test <<<<
```

Hintergrund

Normalerweise brauchen Sie Funktionen nicht zu löschen. Sie werden automatisch gelöscht, wenn PowerShell beendet wird. Möchten Sie eine Funktion ändern, definieren Sie sie einfach erneut.

Funktion mit Schreibschutz versehen

Sie wollen verhindern, dass eine Funktion nachträglich gelöscht oder verändert werden kann.

Lösung

Legen Sie die Funktion mit *Set-Item* direkt im virtuellen Laufwerk *Function:* an und aktivieren Sie mit dem Parameter *-option Constant* den Schreibschutz:

```
PS > Set-Item Function:SichereFunktion { 'Kann nicht geändert werden' } -option Constant
Die Funktion kann anschließend weder geändert noch gelöscht werden:
PS > Function SichereFunktion { 'Neuer Inhalt' }
Die Funktion "SichereFunktion" kann nicht geschrieben werden, da sie schreibgeschützt oder
konstant ist.
Bei Zeile:1 Zeichen:25
+ Function SichereFunktion <<<< { 'Neuer Inhalt' }
PS > del Function:SichereFunktion
Remove-Item : Die Funktion "SichereFunktion" kann nicht entfernt werden, da sie konstant
ist.
Bei Zeile:1 Zeichen:4
+ del <<<< Function:SichereFunktion
```

Hintergrund

Funktionen können auf zwei unterschiedlichen Wegen erstellt werden. Normalerweise verwenden Sie dazu das Schlüsselwort *Function*. Alternativ können Sie die Funktion aber auch direkt mit *Set-Item* im virtuellen Laufwerk *Function:* definieren. Wenn Sie das tun, steht Ihnen mit dem Parameter *-option* die Möglichkeit zur Verfügung, diesen Eintrag mit einem Schreibschutz zu versehen.

Geben Sie hinter *–option* den Wert *ReadOnly* an, wird die Funktion schreibgeschützt, kann aber dennoch mit dem Parameter *–force* überschrieben werden. Wählen Sie dagegen die Einstellung *Constant*, kann diese Funktion während der laufenden PowerShell-Sitzung überhaupt nicht mehr geändert werden. Erst wenn PowerShell beendet wird, werden sämtliche Funktionen gelöscht.

TIPP

Legen Sie die Funktion mit Schreibschutz innerhalb eines Ihrer Profilskripts fest, kann diese Funktion vom Anwender nicht verändert werden.

Wenn Sie allerdings Funktionen mit *Set-Item* anlegen, müssen Sie etwaige Parameter grundsätzlich mit einem *param()*-Block festlegen.

```
PS > Function MwSt($betrag) {
>> $betrag * 0.19
>> }
>>
PS > Set-Item Function:MwSt { param($betrag) $betrag * 0.19 } -option ReadOnly
```

Pipeline-Filter anlegen

Sie möchten eine Funktion innerhalb der Pipeline verwenden und deshalb einen Filter definieren.

Lösung

Gehen Sie so vor, als wollten Sie eine Funktion definieren, aber ersetzen Sie das Schlüsselwort *Function* durch das Schlüsselwort *Filter*. Der folgende Filter *OnlyFiles* kann innerhalb der Pipeline eingesetzt werden, um nur Dateien durch die Pipeline zu lassen und alle übrigen Objekte auszufiltern:

```
PS > Filter OnlyFiles {
>> If ($_.PSIsContainer -eq $false) { $_ }
>> }
>>
```

Filter dürfen direkt in der Pipeline verwendet werden. Die folgende Zeile liefert nur Dateien, aber keine Ordner:

```
PS > dir | OnlyFiles

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias
```

Mode	LastWriteTime	Length	Name
-a---	21.05.2008 09:16	8277	arbeitsblatt.xls
-a---	09.06.2008 12:29	39148	ausgabe.htm
(...)			

Hintergrund

Der Filter im Beispiel macht sich die Eigenschaft *PSIsContainer* zunutze, die bei Datei- und Ordnerobjekten darüber informiert, ob es sich um einen Ordner handelt oder nicht. Ebenso gut hätten Sie aber auch nach anderen Kriterien filtern können, beispielsweise nach dem Objekttyp. Der folgende Filter lässt nur Objekte passieren, die vom Typ *System.IO.DirectoryInfo* sind, also Ordner:

```
PS > filter OnlyFolder {
>> If ($_ -is [System.IO.DirectoryInfo]) { $_ }
>> }
>>
PS > dir | OnlyFolder

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias
```

Mode	LastWriteTime	Length	Name
d----	19.11.2007 21:57		<DIR> Application Data
d----	13.05.2008 12:43		<DIR> Auswertung
d----	30.05.2008 07:16		<DIR> bin
d----	16.11.2007 23:46		<DIR> Bluetooth Software
(...)			

Technisch gesehen sind Filter lediglich Funktionen mit einem *process*-Block. Die folgenden beiden Ansätze sind identisch:

```
Filter OnlyFiles {
  If ($_.PSIsContainer -eq $false) { $_ }
}

Function OnlyFiles {
  process {
    If ($_.PSIsContainer -eq $false) { $_ }
  }
}
```

Sie lesen mehr zu *process*-Blöcken im Abschnitt über pipelinefähige Skripts.

Feststellen, ob ein Fehler aufgetreten ist

Sie möchten herausfinden, ob die letzte Anweisung fehlerfrei ausgeführt wurde.

Lösung

Werten Sie die Variable *\$?* aus. Sie enthält *\$true*, wenn die letzte Anweisung fehlerfrei ausgeführt wurde, andernfalls *\$false*.

```
PS > del "gibtsnicht" -errorAction "SilentlyContinue"
PS > If (!$?) { "Hat nicht geklappt!"; break }; "Alles ok!"
Hat nicht geklappt!
```

Die Fehlerursache findet sich in der Variablen `$error[0]`:

```
PS > "Ups: $error[0]"
Remove-Item : Der Pfad "C:\Users\Tobias\gibtsnicht" kann nicht gefunden werden, da er nicht
vorhanden ist.
Bei Zeile:1 Zeichen:4
+ del <<<< "gibtsnicht" -errorAction "SilentlyContinue"
```

Die Fehlermeldung kann in Ihrem Fall natürlich von diesem Beispiel abweichen und hängt davon ab, welcher Fehler zuletzt innerhalb der PowerShell aufgetreten ist.

Oder setzen Sie einen *try*-Block ein. Die Fehlermeldung steht in `$_` im *catch*-Block zur Verfügung, der zwingend dem *try*-Block folgen muss:

```
PS > try { del "gibtsnicht" -errorAction "Stop" } catch { "Ups: $_" }
Ups: Der Pfad "C:\Users\w7-pc9\gibtsnicht" kann nicht gefunden werden, da er nicht
vorhanden ist.
```

Möchten Sie den Fehlerstatus einer konsolenbasierten Anwendung überprüfen, verwenden Sie die Variable `$lastexitcode`. Sie entspricht dem *ErrorLevel* bei klassischen Batchdateien und enthält den Rückgabewert des externen Programms. Was dieser Rückgabewert im Einzelnen bedeutet, hängt vom Programm ab, das den Rückgabewert liefert. Bei *Ping.exe* bedeutet der Rückgabewert 0 zum Beispiel, dass eine Antwort empfangen wurde, und der Wert 1, dass nichts empfangen wurde:

```
PS > ping.exe -n 1 127.0.0.1

Ping wird ausgeführt für 127.0.0.1 mit 32 Bytes Daten:
Antwort von 127.0.0.1: Bytes=32 Zeit<1ms TTL=128
(...)
PS > $lastexitcode
0
PS > ping.exe -n 1 10.10.10.10

Ping wird ausgeführt für 10.10.10.10 mit 32 Bytes Daten:
Zeitüberschreitung der Anforderung.
(...)
PS > $lastexitcode
1
```

Hintergrund

PowerShell protokolliert sämtliche Fehler in der Variablen `$error`, die also ein Feld ist. Die letzte (aktuellste) Fehlermeldung findet sich am Anfang und Sie rufen diese über `$error[0]` ab. Darüber hinaus signalisiert die Variable `?`, ob in der letzten Anweisung ein Fehler auftrat.

Oft führen die Kürzel zu kryptischen Ausdrücken wie `»(!$?)«`: `»!` steht für das logische »Nicht«. Die Bedingung ist also erfüllt, wenn die Variable `?` nicht den Wert `$true` enthält (und also ein Fehler passiert ist). `Break` sorgt dafür, dass die Zeile nicht weiter ausgeführt wird.

Stammen die Fehler von .NET Framework, kann man diese übersichtlicher mit `try` abfangen. Ein Fehler im `try`-Block führt den folgenden `catch`-Block aus. Auf diese Weise lassen sich auch Fehler abfangen, die von Cmdlets ausgelöst wurden, wenn der Parameter `-ErrorAction` auf `Stop` gesetzt wurde. Andernfalls behandelt das Cmdlet den Fehler intern. Setzen Sie die Variable `$ErrorActionPreference = 'Stop'`, wenn Sie die `ErrorAction`-Einstellung nicht bei jedem Cmdlet einzeln festlegen wollen.

Stammen Fehler aus konsolenbasierten Anwendungen, muss die Variable `$LASTEXITCODE` überprüft werden, weil konsolenbasierte Anwendungen keine .NET-Fehler auslösen. Der Wert in dieser Variablen ist normalerweise 0, wenn der Befehl erfolgreich verlief. Was andere Werte bedeuten, ist von Anwendung zu Anwendung unterschiedlich.

TIPP

Die Farbeinstellungen der PowerShell-Fehlermeldungen finden Sie in `$host.PrivateData`. Die folgenden Zeilen sorgen für rote Fehlermeldungen auf weißem Grund:

```
$host.PrivateData.ErrorForegroundColor = "Red"
$host.PrivateData.ErrorBackgroundColor = "White"
```

Dort finden Sie auch weitere Eigenschaften, mit denen Sie die Farbe von Warn- und Debugmeldungen festlegen.

Jeder Fehler wird in `$error` als sogenannter Error Record mit den Eigenschaften aus Tabelle 6.3 gespeichert. Deshalb können Sie auch gezielte Informationen aus dem Error Record entnehmen:

Eigenschaft	Beschreibung
<i>CategoryInfo</i>	Der Fehler wird kategorisiert nach allgemeiner Kategorie, Aktivität, Grund, Aufrufer und Aufruftyp. Auf diese Weise können ähnliche Fehler unterschiedlicher Herkunft erkannt und gemeinsam behandelt werden.
<i>ErrorDetails</i>	Häufig leer; hier können Entwickler zusätzliche Informationen über den Fehler hinterlegen
<i>Exception</i>	Die .NET-Ausnahme, die den zugrunde liegenden Fehler repräsentiert. Über <i>Exception.Message</i> erhalten Sie die Fehlermeldung.
<i>FullyQualifiedErrorID</i>	Eindeutige spezielle Fehlerkennung, über die Sie den Fehler identifizieren und entsprechende Folgehandlungen auslösen können
<i>InvocationInfo</i>	Liefert Informationen, an welcher Stelle der Fehler ausgelöst wurde, also beispielsweise den Namen eines Skripts sowie die Position im Skript

Tabelle 6.3 Eigenschaften eines Error Records

Eigenschaft	Beschreibung
<i>TargetObject</i>	Das Objekt, mit dem gearbeitet wurde, als der Fehler auftrat. Oft leer oder Text, der dem Argument entsprach, das von einem Cmdlet nicht verarbeitet werden konnte.

Tabelle 6.3 Eigenschaften eines Error Records (*Fortsetzung*)

Dieselben Informationen stehen innerhalb des *catch*-Blocks auch in der Variablen `$_` zur Verfügung:

```
PS > try { del "c:\gibtsnicht" -errorAction "Stop" } catch { '{0}' →
    (Auslöser: {1})' -f $_.Exception.Message, $_.CategoryInfo.Activity }
```

Der Pfad "C:\gibtsnicht" kann nicht gefunden werden, da er nicht vorhanden ist. (Auslöser: Remove-Item)

Fehler in Skripts und Funktionen abfangen

Sie möchten Fehler in Ihren Skripts oder Funktionen abfangen und darauf reagieren, anstatt die eingebauten Fehlermeldungen anzuzeigen.

Lösung

Verwenden Sie die Anweisung *Trap*, wenn Sie Fehler im gesamten Skript oder der gesamten Funktion abfangen wollen, und fangen Sie den Fehler mit der Anweisung *continue* ab. Informationen über den Fehler stehen innerhalb des *trap*-Blocks als Error Record in `$_` zur Verfügung.

```
Trap {
    "Ein Fehler: $_"
    continue
}

"Start"
1/$null
Dir zumsel: -ErrorAction Stop
"Ende"
```

Das Ergebnis sieht so aus:

```
Start
Ein Fehler: Es wurde versucht, durch 0 (null) zu teilen.
Ein Fehler: Das Laufwerk wurde nicht gefunden. Ein Laufwerk mit dem Namen "zumsel" ist
nicht vorhanden.
Ende
```

Setzen Sie einen *try*-Block ein, wenn Sie Fehler in einem bestimmten Bereich abfangen wollen:


```
"Start"
try { 1/$null }
catch { "Ein Fehler: $_" }
try { Dir zumsel: -ErrorAction Stop }
catch { "Ein anderer Fehler: $_" }
"Ende"
```

Das Ergebnis sieht diesmal so aus:

```
Start
Ein Fehler: Es wurde versucht, durch 0 (null) zu teilen.
Ein anderer Fehler: Das Laufwerk wurde nicht gefunden. Ein Laufwerk mit dem Namen "zumsel"
ist nicht vorhanden.
Ende
```

Sowohl Traps als auch *Catch*-Blöcke können auf bestimmte Fehlerarten festgelegt werden, indem man den Typ des Fehlers in eckigen Klammern angibt. Die Trap beziehungsweise der *Catch*-Block reagieren dann nur noch auf die angegebene Fehlerart.

```
Trap [System.DivideByZeroException] {
    "Durch Null geteilt!"
    Continue
}

Trap [System.Management.Automation.ParameterBindingException] {
    "Parameter falsch!"
    Continue
}

1/$null
dir -zumsel
```

Das Ergebnis:

```
Durch Null geteilt!
Parameter falsch!
```

Hintergrund

Alle von .NET Framework ausgelösten Fehler können über einen *trap*-Block oder den *try*-Block mit nachfolgendem *catch*-Block abgefangen werden. Für Cmdlets gilt dies aber nur, wenn die *ErrorAction* auf *Stop* festgelegt ist. Andernfalls behandeln die Cmdlets Fehler intern auf eigene Weise. Mit der folgenden Anweisung kann die Vorgabe für die *ErrorAction* auf *Stop* festgelegt werden, sodass der Parameter *-ErrorAction* nicht für jedes Cmdlet einzeln angegeben werden muss:

```
$ErrorActionPreference = 'Stop'
```

Während eine *Trap* für den gesamten Bereich gilt, also für das gesamte Skript oder die gesamte Funktion, fängt ein *try*-Block Fehler nur in seinem eigenen Skriptblock ab. Tritt in einem *try*-Block ein Fehler auf, wird der Block verlassen und der Code im nachfolgenden *catch*-Block ausgeführt. Der Code im *try*-Block wird also abgebrochen.

Verwenden Sie einen *trap*-Block, so arbeiten Sie im Grunde mit einem *catch*-Block ohne *try*-Block. Es wird also der gesamte Code überwacht. Bei einem Fehler wird der *trap*-Block angesprungen. Abgefangen wird der Fehler jetzt nur, wenn Sie am Ende des *trap*-Blocks die Anweisung *continue* geben. Andernfalls wird der Fehler anschließend an die nächsthöhere Ebene weitergeleitet und führt am Ende dazu, dass PowerShell seine eigene Fehlermeldung ausgibt.

Zwischen *try* und *trap* gibt es einen weiteren Unterschied: Nachdem der Fehler behandelt wurde, setzen beide die Ausführung des restlichen Codes mit der nächsten Anweisung fort, die auf ihrer eigenen Ebene liegt. Bei einem Fehler wird der *try*-Block also komplett abgebrochen, während *trap* mit der nächsten Anweisung nach dem Fehler fortfährt.

Möchten Sie, dass *trap* nicht mit der nächsten Anweisung fortfährt, sondern einige Anweisungen überspringt, müssen Sie den Code in einem untergeordneten Skriptblock unterbringen. Im folgenden Beispiel wird der Code nach dem Fehler nicht mit der nächsten Anweisung fortgesetzt, sondern mit der nächsten Anweisung außerhalb des Skriptblocks, in dem der Fehler passiert ist:

```
Trap {
    "Fehler: $_"
    continue
}

& {
    1/$null
    "Wird nicht mehr ausgeführt"
}
"Wird nach dem Fehler ausgeführt"
```

Sowohl *trap* als auch *try* reagieren nur auf Fehler, die von .NET Framework ausgelöst werden und also normalerweise von PowerShell mit einer roten Fehlermeldung angezeigt werden. Fehler, die von konsolenbasierten Anwendungsprogrammen ausgelöst werden, lassen sich damit nicht abfangen.

TIPP

Sie verwandeln einen *try/catch*-Fehlerhandler in einen globalen *trap*-Fehlerhandler, indem Sie das Konstrukt *try {}* aus dem Code entfernen, *catch* in *trap* umbenennen und am Ende des Fehlerhandlercodes die Anweisung *continue* hinzufügen. Aus optischen Gründen sollte der *trap*-Block an den Anfang des Skripts oder der Funktion verschoben werden.

Aus diesem Code ...

```
try {
    1/$null
}

catch {
    "Fehler $_"
}
```

... wird dieser Code:

```
trap {  
    "Fehler $_"  
    continue  
}  
  
1/$null
```

Eigene Fehler auslösen

Sie wollen in einem Skript oder einer Funktion einen eigenen Fehler auslösen, der dann entweder von PowerShell ausgegeben wird oder sich vom Aufrufer abfangen lässt.

Lösung

Verwenden Sie die Anweisung *Throw*. Die folgende Zeile löst eine Fehlermeldung aus:

```
Throw "Skript XYZ konnte nicht ausgeführt werden."
```

Hintergrund

Die Anweisung *Throw* erzeugt einen neuen .NET-Fehler und unterbricht den aktuellen Code. Wird dieser Fehler nicht abgefangen, erreicht er PowerShell und ergibt eine rote Fehlermeldung.

```
trap {  
    "Ein Fehler: $_"  
    continue  
}  
  
function test {  
    "Beginn"  
    Throw "Unerwartetes Problem in Funktion 'test'"  
    "Ende"  
}  
  
"Rufe Funktion auf."  
test  
"Funktion ist beendet."
```

Dieser Code löst in der Funktion *test* mit *Throw* einen Fehler aus. Der Fehler wird von der *trap* des Skripts aufgefangen und behandelt. Hier wurde also die Information über den Fehler nicht direkt in der Funktion behandelt, sondern vom Skript, in dem die Funktion vorlag. So sieht das Ergebnis aus:

```
Rufe Funktion auf.  
Beginn  
Ein Fehler: Unerwartetes Problem in Funktion 'test'  
Funktion ist beendet.
```

Der Vorteil dieser Konstruktion: Sie benötigen nur einen zentralen Fehlerhandler im Skript, der dann die Fehler sämtlicher Funktionen behandeln kann.

Sie können auf diese Weise auch Fehler gegen andere Fehlertypen und -meldungen austauschen:

```
trap {  
    "Ein Fehler: $_"  
    continue  
}  
  
function test {  
    trap {  
        Throw "Unerwartetes Problem in 'test': $_"  
    }  
    "Beginn"  
    dir zumsel: -ErrorAction Stop  
    "Ende"  
}  
  
"Rufe Funktion auf."  
test  
"Funktion ist beendet."
```

Das Ergebnis:

```
Rufe Funktion auf.  
Beginn  
Ein Fehler: Unerwartetes Problem in 'test': Das Laufwerk wurde nicht gefunden. Ein Laufwerk  
mit dem Namen "zumsel" ist nicht vorhanden.  
Funktion ist beendet.
```

Hier hat die *trap* der Funktion den Fehler erkannt und mit *Throw* eine andere Fehlermeldung weitergereicht und erreichte die *trap* des Skripts. Dort wurde die Fehlermeldung ausgegeben und mit *continue* für erledigt erklärt. Deshalb erscheint keine PowerShell-Fehlermeldung.

Zusammenfassung

Skripts sind Textdateien mit der Erweiterung *.ps1*, die ähnlich funktionieren wie Batchdateien: sie enthalten beliebig viele Zeilen PowerShell-Code, der der Reihe nach abgearbeitet wird. Damit eignen sich Skripts dazu, größere Aufgaben zu formulieren, für die eine einzelne Zeile PowerShell-Code nicht ausreicht. Weil Skripts zudem in Texteditoren verfasst werden, die zeitgemäße Editierfunktionen bieten als die interaktive PowerShell-Konsole, lassen sich längere Codezeilen und besonders Code, der aus mehreren Zeilen besteht, einfacher eingeben.

Damit Skripts ausgeführt werden können, muss einmalig die sogenannte »ExecutionPolicy« konfiguriert werden. Sie bestimmt, unter welchen Umständen PowerShell-Skripts ausgeführt werden dürfen und verbietet anfangs jegliche Ausführung.

Innerhalb von Skripts lassen sich Codeteile als Funktionen kapseln. Funktionen arbeiten ganz ähnlich wie Makros. Sie fassen mehrere Zeilen Code zu einer neuen Aufgabe zusammen und erzeugen daraus einen neuen Befehl. Für den Anwender lassen sich Funktionen damit genauso einsetzen wie Cmdlets.

Funktionen dürfen nicht nur innerhalb von Skripts erzeugt werden. Sie können auch direkt in der interaktiven PowerShell-Konsole eingegeben werden, gelten dann aber nur in der aktuellen PowerShell-Sitzung. Ein besserer Ansatz definiert Funktionen, die häufig genutzt werden sollen, in Skripts. Wird das Skript innerhalb der Konsole »dot-sourced« aufgerufen, also mit einem vorangestellten Punkt, stehen anschließend alle darin definierten Funktionen zur Verfügung. Definiert man Funktionen in einem der speziellen Profilskripts, stehen diese sogar automatisch zur Verfügung, weil Profilskripts bei jedem Start der PowerShell automatisch ausgeführt werden.

Für Skripts und Funktionen gemeinsam gilt, dass man sie mit Parametern und einer Hilfe ausstatten kann. Werden Parameter festgelegt, kann der Anwender ihnen genauso wie bei Cmdlets Zusatzinformationen übergeben. Wird eine Hilfe eingefügt, kann der Anwender zu Funktionen und Skripts über *Get-Help* dieselbe komfortable Hilfe abrufen wie bei Cmdlets.

Kapitel 7

Dateisystem

In diesem Kapitel:

Inhalt eines Ordners auflisten	208
Dateien finden, die einem Kriterium entsprechen	211
Relative Pfadnamen verwenden	213
Relativen Pfad auflösen	214
Prüfen, ob ein Pfad Platzhalter enthält	217
Pfadnamen konstruieren	218
Pfadbestandteile auswerten	219
Systempfad ermitteln	220
Dateien kopieren	224
Ordner kopieren	225
Neue Ordner anlegen	226
Neue Datei anlegen	227
Text in eine Datei schreiben	230
Text aus einer Datei lesen	232
Inhalt einer Textdatei löschen	233

Datei löschen	234
Datei oder Ordner umbenennen	235
Auf erweiterte Dateieigenschaften zugreifen	237
Dateiattribute lesen und ändern	240
Dateien mit einem bestimmten Attribut finden	242
Dateinamen mit speziellen Zeichen verarbeiten	244
Aktuellen Ordner bestimmen (oder setzen)	246
Pfadnamen auflösen	248
Veränderungen an Ordnerinhalten per Snapshot auswerten	250
Echtzeit-Überwachung von Änderungen an Ordnern	252
Echtzeit-Überwachung von Änderungen an Dateien	257
Ungültige Dateinamen ermitteln	260
Zusammenfassung	260

Das Dateisystem bildet einen besonderen Schwerpunkt der Arbeit eines Administrators. Hier lagern alle Dateien und Ordner und häufig müssen Backups durchgeführt oder Logbuchdateien ausgewertet werden. In der Praxis werden oft zusätzliche Anforderungen gestellt: Man möchte wissen, ob sich ein Ordnerinhalt geändert hat (zum Beispiel, weil es ein Dropverzeichnis eines Diensts ist), Foto-Bibliotheken sollen einheitlich umbenannt werden oder die Metadaten (erweiterte Eigenschaften) von Dateien müssen ausgelesen werden.

Inhalt eines Ordners auflisten

Sie möchten den Inhalt eines Ordners auflisten und dabei gegebenenfalls vorhandene Unterordner einschließen.

Lösung

Get-ChildItem listet den Inhalt des aktuellen Ordners auf. Für dieses Cmdlet existieren zwei »historische Aliasnamen«, *dir* und *ls*, die anstelle des etwas sperrigen Cmdlet-Namens *Get-ChildItem* verwendet werden dürfen.

So listen Sie den Inhalt des aktuellen Ordners auf:

```
PS > Get-ChildItem
```

Welches der aktuelle Ordner ist, verrät *Get-Location*. *Set-Location* (historischer Alias: *cd*) setzt den aktuellen Ordner auf einen anderen Ort fest.

Eindeutiger ist die Angabe eines festen Pfadnamens. Die folgende Zeile listet den Ordner *c:\windows* auf:

```
PS > Get-ChildItem c:\windows
```

Anstelle den Pfadnamen fest vorzugeben, kann dieser auch aus einer Variable oder Umgebungsvariable stammen. Die folgende Zeile listet immer den Windows-Ordner auf, ganz gleich, ob sich dieser am Standardort *c:\windows* oder vielleicht an anderer Stelle unter anderem Namen befindet. Der aktuelle Pfad wird aus der Umgebungsvariable *windir* gelesen:

```
PS > Get-ChildItem $env:windir
```

Auch PowerShell hält einige vordefinierte Variablen vor. Mit der Variablen *\$home* sprechen Sie immer die Wurzel Ihres Benutzerprofilordners an:

```
PS > Get-ChildItem $home
```

Möchten Sie nur Dateien finden, die einem bestimmten Namenskriterium entsprechen, verwenden Sie Platzhalterzeichen. Die folgende Zeile liefert alle Logbuchdateien im *Windows*-Ordner:

```
PS > Get-ChildItem $env:windir *.log
```

Möchten Sie Dateien rekursiv auch in allen Unterordnern finden, fügen Sie die Parameter *-recurse* hinzu:

```
PS > Get-ChildItem $env:windir -filter *.log -recurse
```

Die rekursive Suche kann lange dauern. Dabei können auch Fehlermeldungen erscheinen, die die Suche aber nicht abbrechen und von Unterordnern stammen, auf die Sie keine Zugriffsrechte haben. Um solche Fehler zu unterdrücken, fügen Sie bei rekursiven Dateisuchen besser den Parameter *-ErrorAction SilentlyContinue* oder *-ea 0* hinzu.

Hintergrund

Get-ChildItem trägt einen sehr abstrakten Namen, weil PowerShell seine Laufwerke nicht auf das Dateisystem beschränkt. Tatsächlich können PowerShell-Laufwerke auch andere Informationsspeicher abbilden und *Get-ChildItem* muss auch dort den Inhalt von Containern auflisten können. Da PowerShell die Elemente auf einem Laufwerk als »Item« bezeichnet, ist der Name *Get-ChildItem* eigentlich sehr treffend.

In der Praxis spielen die sehr viel kürzeren Aliasnamen *dir* und *ls* eine große Rolle. Sie rufen intern ebenfalls *Get-ChildItem* auf, was erklärt, dass die Parameter von *dir* und *ls* nicht den historischen Parametern dieser Befehle entsprechen. Stattdessen müssen Sie auch bei *dir* und *ls* die Parameter von *Get-ChildItem* einsetzen.

Weil man im Alltag insbesondere die Dateisystem-Cmdlets häufig ohne großes Nachdenken und ohne Angabe von Parameternamen einsetzt, kann es zu Missverständnissen und unerwarteten Resultaten kommen. Die folgende Zeile liefert beispielsweise nicht wie erwartet die *.log*-Dateien rekursiv aus allen Ordnern und Unterordnern im Windows-Ordner:

```
PS > Get-ChildItem $env:windir\*.log -recurse
```

Mit benannten Parametern sieht dieser Aufruf folgendermaßen aus:

```
PS > Get-ChildItem -Path $env:windir\*.log -recurse
```

Get-ChildItem liefert alles, was in *Path* angegeben ist, also in diesem Fall alle Dateien, die auf *.log* enden. Der Parameter *-recurse* kann nicht wirksam werden, weil das Cmdlet keine Unterordner mehr liefert, es sei denn, ihr Name endet zufällig ebenfalls mit *.log*.

Wenn Sie also Rekursion einsetzen wollen, dürfen Sie Unterordner nicht von vornherein mit *-path* ausschließen. Formulieren Sie Ihre Einschränkung mit dem separaten Parameter *-filter*:

```
PS > Get-ChildItem -Path $env:windir -filter *.log -recurse # benannte Parameter
PS > Get-ChildItem $env:windir *.log -recurse             # positionale Parameter
```

Die Parameter *-include* und *-filter* scheinen auf den ersten Blick dasselbe zu tun, funktionieren aber fundamental verschieden, was deutlich wird, wenn man mit *Measure-Command* die Ausführungsgeschwindigkeit analysiert:

```
PS > Measure-Command { Get-ChildItem $env:windir -include *.log -recurse -ea >>
    SilentlyContinue }
(...)
TotalSeconds      : 57,9363424
(...)

PS > Measure-Command { Get-ChildItem $env:windir -filter *.log -recurse -ea >>
    SilentlyContinue }
(...)
TotalSeconds      : 9,7523121
(...)
```

Der Parameter *-include* ist fünf- bis zehnmal langsamer als *-filter*. Grund: *-include* und *-exclude* sind clientseitige Filter. *Get-ChildItem* ruft also zuerst alle Ordnerinhalte ab und erst danach filtern diese Parameter die Ergebnisse. Das kostet Zeit, ist aber dafür ein sehr robuster Mechanismus, der für alle Laufwerksprovider funktioniert. *-filter* wendet das Suchkriterium dagegen bereits auf die Suche im Dateisystem an (serverseitige Filterung). Das ist schneller, weil weniger Daten gelesen werden müssen, wird aber nur von manchen Providern unterstützt. Im Dateisystem sollten Sie also immer *-filter* vorziehen, es sei denn, Sie wollen auf einen besonderen Vorzug des Parameters *-include* zugreifen: Er akzeptiert kommasepariert mehrere Filterkriterien, kann also gleichzeitig nach mehreren Dateitypen suchen:

```
PS > Get-ChildItem $home -recurse -include *.doc, *.bmp
```

Normalerweise liefert *Get-ChildItem* die Inhalte eines Ordners als Objekte. Interessieren Sie sich nur für die Namen der Ordnerinhalte, verwenden Sie die Option *-name*:

```
PS > Get-ChildItem -name
```

Get-ChildItem akzeptiert mehrere Ordner, wenn sie mit Kommata getrennt angegeben werden. Die folgende Anweisung listet sämtliche Textdateien sowohl aus dem Windows-Ordner als auch aus Ihrem Profil auf:

```
PS > Get-ChildItem $env:windir, $home -filter *.ps1 -recurse -ea SilentlyContinue
```

Durch Angabe der *ErrorAction SilentlyContinue* werden dabei Fehlermeldungen unterdrückt, falls *Get-ChildItem* mangels Zugriffsrechten einzelne Ordnerinhalte nicht lesen kann.

Dateien finden, die einem Kriterium entsprechen

Sie möchten alle Dateien finden, die nach einem bestimmten Datum geändert oder angelegt wurden oder einem anderen Kriterium entsprechen.

Lösung

Möchten Sie Dateien aufgrund ihres Namens filtern, verwenden Sie den Parameter *-path* und die darin unterstützten Platzhalterzeichen. Die folgende Zeile liefert alle Textdateien, die mit »A« beginnen:

```
PS > Get-ChildItem A*.txt
```

Sie können auf diese Weise alle ausführbaren Dateien im Windows-Ordner finden, deren Name auf »32« endet:

```
PS > Get-ChildItem $env:windir -filter '*32.exe'  
PS > Get-ChildItem $env:windir '*32.exe'
```

Enthält ein Ordnername Zeichen, die von PowerShell als Platzhalterzeichen interpretiert werden könnten, greifen Sie anstelle von *-path* zum Parameter *-literalPath*:

```
PS > Get-ChildItem c:\logs[neu]
```

Alle sonstigen Dateierkmale überprüfen Sie, indem Sie das Ergebnis von *Get-ChildItem* an einen Filter (*Where-Object*) weiterleiten, und prüfen Sie darin jeweils, ob die von *Get-ChildItem* gelieferten Dateien Ihren Kriterien entsprechen. Auf diese Weise lassen sich zum Beispiel regu-

läre Ausdrücke verwenden, um Textmuster zu finden. Die folgende Zeile findet alle *.log*-Dateien, deren Name mit »KB« beginnt und wo danach drei bis sechs Zahlen folgen.

```
PS > Get-ChildItem $env:windir | Where-Object { $_.name -match '^KB\d{3,6}\.log$' }
```

Diese Anweisung findet alle Dateien in Ihrem Benutzerprofil, die innerhalb der letzten drei Tage verändert wurden.

```
PS > Get-ChildItem $home | Where-Object { $_.LastWriteTime -gt →  
(Get-Date).AddDays(-3) }
```

Möchten Sie auch die Unterordner einschließen, fügen Sie den Parameter *-recurse* hinzu:

```
PS > Get-ChildItem $home -recurse | Where-Object { $_.LastWriteTime -gt →  
(Get-Date).AddDays(-3) }
```

Diese Zeile listet alle Dateien im aktuellen Ordner auf, die innerhalb der letzten zehn Tage angelegt wurden:

```
PS > Get-ChildItem | ? { $_.CreationTime -gt (Get-Date).AddDays(-10) }
```

Und diese Zeile findet alle Dateien im Windows-Ordner und seinen Unterordnern, die größer sind als 1 Mbyte:

```
PS > Get-ChildItem $env:windir -recurse -ea 0 | Where-Object { $_.Length -gt 1MB }
```

Erstaunlicherweise unterstützt der Parameter *-filter* nicht mehrere Filter parallel, sodass Sie mit *Get-ChildItem* nicht nach mehreren Dateierweiterungen gleichzeitig suchen können. Verwenden Sie hierfür den langsameren aber flexibleren Parameter *-include*. Die nächste Zeile findet rekursiv im gesamten Windows-Ordner sämtliche *.wav*-Audio- und *.bmp*-Bilddateien:

```
PS > Get-ChildItem c:\windows -include *.bmp,*.wav -Recurse -ea 0
```

Hintergrund

Um nur bestimmte Dateien zu finden, schauen Sie sich zuerst die Parameter von *Get-ChildItem* an, denn diese arbeiten am effektivsten. Mit *-filter* lassen sich Platzhalterzeichen einsetzen, um nach Dateinamen zu filtern.

Alle übrigen Filterwünsche löst der Cmdlet-Filter *Where-Object*. Man übergibt ihm in geschweiften Klammern die Anweisung, die für jedes Element der Pipeline ausgeführt werden soll. Nur wenn diese Anweisung das Ergebnis *\$true* liefert, wird das Objekt durch den Filter gelassen.

Das jeweilige Objekt findet sich innerhalb der geschweiften Klammern in der Variablen `$_`. Sie könnten nun also prüfen, ob eine Eigenschaft dieses Objekts Ihren Kriterien entspricht oder nicht. Interessieren Sie sich zum Beispiel nur für Dateien, aber nicht für Ordner, greifen Sie auf die Eigenschaft `PSIsContainer` zu. Die nächste Anweisung liefert nur Dateien, aber keine Unterordner:

```
PS > Get-ChildItem $env:windir | Where-Object { $_.PSIsContainer -eq $false }
```

An diesem Beispiel sieht man, dass viele Wege nach Rom führen können. Alle folgenden Anweisungen liefern dasselbe Resultat:

```
PS > Get-ChildItem | Where-Object { -not $_.PSIsContainer }  
PS > Get-ChildItem | Where-Object { -not (($_ -is [System.IO.DirectoryInfo]) ) }  
PS > Get-ChildItem | Where-Object { -not ( $_.Attributes -band [IO.FileAttributes]"Directory") }  
PS > Get-ChildItem | Where-Object { -not ($_.Attributes -band 16) }
```

Welche Eigenschaften eines Datei- oder Ordnerobjekts Ihnen für Filterungen zur Verfügung stehen, sehen Sie, wenn Sie die Ergebnisse von `Get-ChildItem` entweder an `Select-Object *` oder an `Get-Member` weiterleiten:

```
PS > Get-ChildItem $env:windir | Select-Object *  
PS > Get-ChildItem $env:windir | Get-Member
```

Relative Pfadnamen verwenden

Sie möchten eine Datei oder einen Ordner relativ zur aktuellen Ordnerposition angeben.

Lösung

Verwenden Sie im Pfadnamen die Sonderzeichen aus Tabelle 7.1 auf Seite 214. Möchten Sie zum Beispiel im übergeordneten Ordner einen neuen Ordner namens *Test* anlegen, gehen Sie so vor:

```
PS > md ..\Test
```

Die Anweisung schlägt fehl, wenn Sie im übergeordneten Ordner keine Schreibberechtigungen haben oder es keinen übergeordneten Ordner gibt.

Hintergrund

Orte im Dateisystem dürfen relativ zur augenblicklichen Position angegeben werden. So wechseln Sie mit `»..«` in den übergeordneten Ordner:

```
PS > cd ..
```

»\« wechselt in das Wurzelverzeichnis (oberstes Verzeichnis) des aktuellen Laufwerks:

```
PS > cd \
PS > Get-Location
Path
----
C:\
```

Die Anweisung »\« ist auch innerhalb von Pfadnamen von großer Bedeutung. So listet die folgende Anweisung das Wurzelverzeichnis des Laufwerks C: auf:

```
PS > Get-ChildItem c:\
```

Ohne »\« würde *Get-ChildItem* dagegen den aktuellen Ordner im Laufwerk C: auflisten:

```
PS > cd $home
PS > Get-ChildItem c:\
PS > Get-ChildItem c:
```

Das Zeichen ».« repräsentiert den aktuellen Ordner. Mit der folgenden Anweisung beauftragen Sie den Explorer, den aktuellen Ordner Ihrer PowerShell-Konsole anzuzeigen:

```
PS > explorer .
```

Das Zeichen »~« schließlich steht für Ihr Basisverzeichnis:

```
PS > cd ~
```

Platzhalter	Beschreibung
.	Repräsentiert den aktuellen Ordner
..	Repräsentiert den übergeordneten Ordner
\	Steht für den Stammordner des aktuellen Laufwerks
~	Repräsentiert das Basisverzeichnis, also die Voreinstellung, wenn PowerShell startet

Tabelle 7.1 Platzhalter für relative Pfadangaben

Relativen Pfad auflösen

Sie wollen einen relativen Pfad in einen oder mehrere absolute Pfadnamen auflösen. Beispielsweise möchten Sie alle Textdateien in einem Ordner mit dem Editor öffnen.

Lösung

Verwenden Sie *Resolve-Path*, um einen relativen Pfad in einen absoluten Pfad umzuwandeln:

```
PS > Resolve-Path .  
Path  
----  
C:\Users\Tobias  
  
PS > Resolve-Path ..\Test  
Resolve-Path : Der Pfad "C:\Users\Test" kann nicht gefunden werden, da er nicht vorhanden ist.  
Bei Zeile:1 Zeichen:13  
+ Resolve-Path <<<< ..\Test  
PS > Resolve-Path *.txt  
Path  
----  
C:\Users\Tobias\ausgabe.txt  
C:\Users\Tobias\datei.txt  
C:\Users\Tobias\logbuch.txt  
  
PS > Resolve-Path c:\win*\sy*\win*s[p-v]o*1  
Path  
----  
C:\Windows\System32\WindowsPowerShell
```

Resolve-Path funktioniert allerdings nur, wenn der relative Pfad tatsächlich vorhanden ist. Möchten Sie auch hypothetische (nicht wirklich vorhandene) relative Pfade auflösen, greifen Sie auf die Methode *GetFullPath()* aus .NET Framework zurück:

```
PS > [Environment]::CurrentDirectory = (Get-Item (Get-Location)).FullName  
PS > [System.IO.Path]::GetFullPath("..\Werner\Test\Unterordner\Nicht vorhanden")  
C:\Users\Werner\Test\Unterordner\Nicht vorhanden
```

Hintergrund

Resolve-Path wandelt nicht nur den von Ihnen angegebenen relativen Pfad in einen absoluten Pfad um, sondern überprüft gleichzeitig, ob der resultierende absolute Pfad auch existiert. Mit *Resolve-Path* lassen sich also nur tatsächlich vorhandene relative Pfade in absolute Pfade umwandeln:

```
PS > Resolve-Path *.txt  
Path  
----  
C:\Users\Tobias\ausgabe.txt  
C:\Users\Tobias\logbuch.txt  
C:\Users\Tobias\neue datei.txt
```

Eine Funktion könnte sich das zunutze machen, um mehrere Dateien gleichzeitig in den Windows Editor (oder jedes andere geeignete Programm) zu laden:

```
Function Open-File{
    Param(
        [Parameter(Mandatory=$true)]
        $pfad
    )
    $pfade = Resolve-Path $pfad -ea SilentlyContinue
    if ($?) {
        $pfade | ForEach-Object { notepad.exe $_ }
    } else {
        "Leider entsprach keine Datei Ihrem Kriterium $pfad."
    }
}

PS > Open-File *.txt
```

Resolve-Path kann auch für ungewöhnliche Aufgaben eingesetzt werden. Möchten Sie zum Beispiel alle *.dll*-Dateien finden, die sich genau zwei Ordnerstufen unterhalb des aktuellen Ordners befinden, könnten Sie so vorgehen:

```
PS > Resolve-Path $env:windir\*\*\*.dll -ea SilentlyContinue
Path
----
C:\Windows\ADAM\de\ADSchemaAnalyzer.resources.dll
C:\Windows\Branding\Basebrd\basebrd.dll
C:\Windows\Branding\ShellBrd\shellbrd.dll
C:\Windows\ehome\CreateDisc\CreateDisc.dll
C:\Windows\ehome\CreateDisc\SBEServerPS.dll
C:\Windows\ehome\CreateDisc\SonicMCEBurnEngine.dll
(...)
```

Die nächste Zeile würde den Inhalt des Desktops für alle lokalen Benutzer auflisten – sofern Sie dafür ausreichende Berechtigungen besitzen:

```
PS > Dir c:\users\*\Desktop\* -ea 0
```

Resolve-Path ist ein »blockierender« Befehl, der sein Ergebnis nicht in Echtzeit liefert, sondern erst, wenn die gesamte Anweisung verarbeitet ist. Es kann daher einige Zeit dauern, bis der Befehl Resultate liefert.

Möchten Sie beliebige – auch nicht vorhandene – relative Pfade in absolute Pfade verwandeln, greifen Sie direkt auf .NET Framework zurück. Verwenden Sie die Methode *GetFullPath()*. Damit diese Methode Ihren relativen Pfad korrekt in einen absoluten Pfad umwandeln kann, muss .NET Framework allerdings den aktuellen Pfad kennen, den Sie innerhalb von PowerShell verwenden. Vor dem Aufruf von *GetFullPath()* stellen Sie deshalb den aktuellen PowerShell-Pfad mit *CurrentDirectory* auch in .NET Framework ein.

Weil .NET Framework die intern von PowerShell verwendeten *PSDrives* (virtuelle Laufwerke) nicht kennt, müssen Sie den internen aktuellen PowerShell-Pfad, den *Get-Location* liefert, zuerst in einen allgemein gültigen Pfad umwandeln. Dazu sprechen Sie mit *Get-Item* den internen PowerShell-Pfad an und ermitteln dann den echten allgemeinen Pfad über die Eigenschaft *FullName*:

```
PS > [Environment]::CurrentDirectory = (Get-Item (Get-Location)).FullName
PS > [System.IO.Path]::GetFullPath("..\Werner\Test\Unterordner\Nicht vorhanden")
C:\Users\Werner\Test\Unterordner\Nicht vorhanden
```

Allerdings unterstützt *GetFullPath()* (verständlicherweise) keine Platzhalterzeichen wie »*«.

Prüfen, ob ein Pfad Platzhalter enthält

Sie möchten wissen, ob eine Pfadangabe absolut ist oder Platzhalterzeichen enthält.

Lösung

Verwenden Sie die Methode *ContainsWildcardCharacters()* von .NET Framework:

```
PS > $pfad1 = "Absoluter Pfad"
PS > $pfad2 = "*.txt"
PS > [Management.Automation.WildcardPattern]::ContainsWildcardCharacters($pfad1)
False
PS > [Management.Automation.WildcardPattern]::ContainsWildcardCharacters($pfad2)
True
```

Hintergrund

ContainsWildcardCharacters() liefert *\$true* zurück, wenn der angegebene Pfad Platzhalterzeichen enthält, sonst *\$false*. Möchten Sie zusätzlich überprüfen, ob es sich um einen relativen Pfad handelt, verwenden Sie *Split-Path* mit dem Parameter *-isAbsolute*.

```
PS > [Management.Automation.WildcardPattern]::ContainsWildcardCharacters("c:\testordner")
False
PS > [Management.Automation.WildcardPattern]::ContainsWildcardCharacters("..\test.txt")
False
PS > [Management.Automation.WildcardPattern]::ContainsWildcardCharacters("*.txt")
True
PS > Split-Path "c:\testordner" -isAbsolute
True
PS > Split-Path "..\test.txt" -isAbsolute
False
PS > Split-Path "*.txt" -isAbsolute
False
```

Möchten Sie zum Beispiel feststellen, ob eine bestimmte vom Anwender angegebene Datei existiert, muss zunächst überprüft werden, ob der Anwender im Dateinamen Platzhalterzeichen verwendet hat.

```
Function FileExists([string]$pfad=$(Throw 'Dateiname angeben!')) {  
    if ([Management.Automation.WildcardPattern]::ContainsWildcardCharacters($pfad)) {  
        # Platzhalter vorhanden  
        # Pfadnamen auflösen  
        Resolve-Path $pfad | ForEach-Object { Test-Path $_ }  
    } else {  
        Test-Path $path  
    }  
}
```

```
PS > FileExists z*
```

Pfadnamen konstruieren

Sie möchten aus verschiedenen Einzelinformationen einen gültigen Pfadnamen konstruieren.

Lösung

Verwenden Sie *Join-Path*, um die Pfadbestandteile korrekt miteinander zu verbinden:

```
PS > $teil1 = [Environment]::GetFolderPath("Desktop")  
PS > $teil2 = "Klick mich an!.lnk"  
PS > $pfad = Join-Path $teil1 $teil2  
PS > $pfad  
C:\Users\Tobias\Desktop\Klick mich an!.lnk
```

Hintergrund

Natürlich können Sie Pfade aus Einzeltexten auch selbst zusammensetzen. Dann allerdings sind Sie selbst verantwortlich dafür, die einzelnen Bestandteile mit »\«-Zeichen korrekt zusammenzufügen:

```
PS > $teil1 = [Environment]::GetFolderPath("Desktop")  
PS > $teil2 = "Klick mich an!.lnk"  
PS > $pfad = $teil1 + '\' + $teil2  
PS > $pfad  
C:\Users\Tobias\Desktop\Klick mich an!.lnk
```

Diese Arbeit nimmt Ihnen *Join-Path* zuverlässig ab, denn es erkennt automatisch, ob bereits »\«-Zeichen vorhanden sind oder nicht:

```
PS > $teil1 = [Environment]::GetFolderPath("Desktop")
PS > $teil2 = "\Klick mich an!.lnk"
PS > $pfad = Join-Path $teil1 $teil2
PS > $pfad
C:\Users\Tobias\Desktop\Klick mich an!.lnk
```

Pfadbestandteile auswerten

Sie wollen einen Pfad in seine Bestandteile zerlegen oder einzelne Informationen wie zum Beispiel den Pfadnamen des übergeordneten Ordners erfahren.

Lösung

Verwenden Sie *Split-Path*, um einen Pfad in seine gebräuchlichsten Bestandteile zu zerlegen:

```
PS > $pfad = $home
PS > Split-Path $pfad                # übergeordneter Ordner
C:\Users
PS > Split-Path $pfad -parent        # übergeordneter Ordner
C:\Users
PS > Split-Path $pfad -leaf          # letztes Element
Tobias
PS > Split-Path $pfad -noQualifier    # ohne Laufwerk
\Users\Tobias
PS > Split-Path $pfad -qualifier      # nur Laufwerk
C:
```

Hintergrund

Split-Path zerlegt einen Pfad in seine gebräuchlichsten Bestandteile. Geben Sie keinen besonderen Parameter an, verwendet *Split-Path* den Parameter *-parent* und liefert dann den übergeordneten Ordner.

Verwenden Sie relative Pfadnamen, können Sie mit dem Parameter *-resolve* dafür sorgen, dass der relative Pfadname automatisch in einen oder mehrere absolute Pfadnamen umgewandelt wird. Die folgende Zeile liefert zum Beispiel die Namen sämtlicher Logdateien im *Windows*-Ordner:

```
PS > Split-Path $env:windir\*.log -leaf -resolve
DtcInstall.log
GRLP.LOG
KB925528.LOG
KB925902.LOG
(...)
```

Dasselbe Ergebnis hätte allerdings auch *Get-ChildItem* mit dem Parameter *-name* geliefert:

```
PS > Get-ChildItem $env:windir\*.log -name
```

Intern greift *Split-Path* auf die .NET-Klasse *System.IO.Path* zurück, die noch einige zusätzliche Funktionalitäten zu bieten hat, die nicht über *Split-Path* zugänglich sind:

PowerShell	Entsprechende .NET-Methode
<i>Join-Path \$teil1 \$teil2</i>	<i>[System.IO.Path]::Combine(\$teil1,\$teil2)</i>
<i>Split-Path \$home</i>	<i>[System.IO.Path]::GetDirectoryName(\$home)</i>
–	<i>[System.IO.Path]::GetExtension("c:\autoexec.bat")</i>
–	<i>[System.IO.Path]::ChangeExtension("c:\autoexec.bat", "bak")</i>
<i>Split-Path \$home –leaf</i>	<i>[System.IO.Path]::GetFileName(\$home)</i>
–	<i>[System.IO.Path]::GetFileNameWithoutExtension("c:\autoexec.bat")</i>
<i>Resolve-Path "..\Test"</i>	<i>[System.IO.Path]::GetFullPath(".. \Test")</i>
–	<i>[System.IO.Path]::GetInvalidFileNameChars()</i>
–	<i>[System.IO.Path]::GetInvalidPathChars()</i>
<i>Split-Path \$home –qualifier</i>	<i>[System.IO.Path]::GetPathRoot(\$home)</i>
–	<i>[System.IO.Path]::GetRandomFileName()</i>
–	<i>[System.IO.Path]::GetTempFileName()</i>
–	<i>[System.IO.Path]::HasExtension(\$home)</i>
<i>Split-Path \$home –isAbsolute</i>	<i>[System.IO.Path]::IsPathRooted(\$home)</i>

Tabelle 7.2 PowerShell-Cmdlets und zugrunde liegende .NET-Methoden

Pfadnamen können auch über Textsplitoperationen extrahiert werden. Die folgenden Zeilen demonstrieren das:

```
PS > # Laufwerk:
PS > ('c:\ordner\unterordner\datei.txt' -split '\\')[0]
PS > # Datei:
PS > ('c:\ordner\unterordner\datei.txt' -split '\\')[-1]
PS > # Extension:
PS > ('c:\ordner\unterordner\datei.txt' -split '\.')[1]
PS > # Datei ohne Extension:
PS > (('c:\ordner\unterordner\datei.txt' -split '\\')[-1] -split '\.')[0]
PS > # übergeordneter Ordner
PS > ('c:\ordner\unterordner\datei.txt' -split '\\')[-2]
```

Systempfad ermitteln

Sie möchten herausfinden, an welchem Ort ein bestimmter Systemordner auf Ihrem PC gespeichert ist.

Lösung

Die Orte vieler wichtiger Ordner finden sich in automatischen PowerShell-Variablen. Die automatische Variable *\$home* liefert zum Beispiel immer den Ordnerpfad Ihres Stammverzeichnisses:

```
PS > $home
C:\Users\w7-pc9
```

Viele Systemordner stehen Ihnen auch über Umgebungsvariablen zur Verfügung. Die Umgebungsvariable *windir* liefert zum Beispiel den Pfad zu Ihrem Windows-Ordner:

```
PS > $env:windir
C:\Windows
```

Sämtliche Umgebungsvariablen finden Sie auf dem Laufwerk »env:«:

```
PS > Get-ChildItem env:
```

Über .NET Framework und den Typ *Environment* erhalten Sie die Ordnerpfade vieler weiterer Systemordner. Den Ordnerpfad des Startmenüs finden Sie beispielsweise so heraus:

```
PS > [Environment]::GetFolderPath("StartMenu")
C:\Users\Tobias\AppData\Roaming\Microsoft\Windows\Start Menu
PS > Get-Childitem ([Environment]::GetFolderPath("StartMenu"))
```

Hintergrund

PowerShell definiert Variablen, die wichtige Ordnerpfade liefern können. Der Ordner *\$pshome* liefert beispielsweise immer den internen PowerShell-Systempfad, wo Sie die technischen Einzelteile von PowerShell finden.

Variable	Inhalt
<i>\$home</i>	Wurzelverzeichnis Ihres persönlichen Profils. Hier haben Sie immer Schreibberechtigungen.
<i>\$pshome</i>	Wurzelverzeichnis von PowerShell. Hier finden Sie zum Beispiel die <i>ps1xml</i> -Formatdateien für das Extended Type System (ETS).
<i>\$pwd</i>	Aktueller Ordner. Entspricht dem Ergebnis von <i>Get-Location</i> .
<i>\$profile</i>	Pfad zum Autostartskript. <i>\$profile</i> besitzt weitere Eigenschaften wie beispielsweise <i>\$profile.AllUsersAllHosts</i> , die die Pfadnamen der übrigen Profilskripts liefern.

Tabelle 7.3 Vordefinierte PowerShell-Variablen mit Pfadnamen wichtiger Ordner

Darüber hinaus erlaubt PowerShell Zugriff auf sämtliche Umgebungsvariablen, die Sie sich im Laufwerk *env*: ansehen und mit *\$env:NAME* abrufen können.

Umgebungsvariable	Ort
<i>ProgramData</i>	Ordner für Programm-Informationen
<i>Tmp, Temp</i>	Ordner für temporäre Informationen. In diesem Ordner haben Sie grundsätzlich Schreibberechtigungen. Sie sind selbst dafür verantwortlich, die hier vorübergehend gespeicherten Informationen nach Gebrauch wieder zu löschen.
<i>LocalAppData</i>	Anwendungsdaten im lokalen Profil
<i>Public</i>	Öffentlicher Ordner. Alle lokalen Anwender haben gemeinsamen Lese- und Schreibzugriff auf diesen Ordner und können hier untereinander Daten austauschen.
<i>Windir</i>	<i>Windows</i> -Ordner
<i>CommonProgramFiles</i>	Öffentlicher Ordner für allgemein gebräuchliche Programmdaten
<i>ProgramFiles</i>	Schreibgeschützter Ordner, in dem sich installierte Programme befinden
<i>UserProfile</i>	Wurzelverzeichnis Ihres Benutzerprofils
<i>HomeDrive</i>	Laufwerksbuchstabe des Standardlaufwerks
<i>AppData</i>	Ordner für Anwendungsdaten, die in Roaming Profiles zentral gespeichert werden können
<i>AllUsersProfile</i>	Wurzelverzeichnis aller Benutzerprofile

Tabelle 7.4 Umgebungsvariablen mit wichtigen Systemordnern

Über die .NET-Klasse *Environment* und die statische Methode *GetFolderPath()* erhalten Sie die Pfade vieler weiterer wichtiger Systemordner. Dazu geben Sie *GetFolderPath()* den Kurznamen des gewünschten Systemordners an. Den Ordner Ihres *Dokumente*-Ordners mit all Ihren persönlichen Daten ermitteln Sie zum Beispiel so:

```
PS > [Environment]::GetFolderPath("MyDocuments")
C:\Users\Tobias\Documents
```

So finden Sie heraus, welche Kurznamen *GetFolderPath()* sonst noch unterstützt:

```
PS > [System.Enum]::GetNames([System.Environment+SpecialFolder])
Desktop
Programs
Personal
(...)
```

Eine Übersicht sämtlicher Ordner, die *GetFolderPath()* liefern kann, ermitteln Sie folgendermaßen:

```
PS > [System.Enum]::GetNames([System.Environment+SpecialFolder]) | -->
    ForEach-Object { "{0,30} = {1,-30}" -f $_, ([Environment]::GetFolderPath($_)) }
    Desktop = C:\Users\Tobias\Desktop
    Personal = C:\Users\Tobias\Documents
    MyDocuments = C:\Users\Tobias\Documents
    Favorites = C:\Users\Tobias\Favorites
    Recent = C:\Users\Tobias\AppData\Roaming\Microsoft\Windows\Recent
    SendTo = C:\Users\Tobias\AppData\Roaming\Microsoft\Windows\SendTo
    (...)
```

Kurzname	Ordner
<i>Desktop</i>	Der Pfadname des Desktops. Er enthält alle privaten Dateien, die Sie auf dem Desktop sehen.
<i>Programs</i>	<i>Programme</i> -Menü im Startmenü
<i>MyDocuments</i>	Ihr persönlicher <i>Dokumente</i> -Ordner
<i>Favorites</i>	Ihre Internet-Favoriten
<i>Startup</i>	Autostart-Programmgruppe
<i>Recent</i>	Ihre zuletzt verwendeten Programme
<i>SendTo</i>	Ihr <i>Senden an</i> -Menü
<i>StartMenu</i>	Wurzelverzeichnis Ihres Startmenüs
<i>MyMusic</i>	Ihr <i>Musik</i> -Ordner
<i>DesktopDirectory</i>	Ihr <i>Desktop</i> -Ordner
<i>MyComputer</i>	(kein gültiger Pfadname)
<i>Templates</i>	Vorlagen für neue Dokumente
<i>ApplicationData</i>	Allgemeine Anwendungsdaten
<i>LocalApplicationData</i>	Anwendungsdaten für den lokalen Computer
<i>InternetCache</i>	Ordner für vorübergehend gespeicherte Internetinhalte
<i>Cookies</i>	Ordner für Internet-Cookies
<i>History</i>	Ordner mit den Verlaufsdaten für Ihren Browser
<i>CommonApplicationData</i>	Lokal gespeicherte Anwendungsdaten für alle Benutzer
<i>System</i>	Pfadname des Systemordners im <i>Windows</i> -Ordner
<i>ProgramFiles</i>	Ordner mit allen installierten Programmen
<i>MyPictures</i>	Ihr <i>Bilder</i> -Ordner
<i>CommonProgramFiles</i>	Allgemeine Programmdateien installierter Programme

Tabelle 7.5 Schlüsselbegriffe für die *GetFolderPath()*-Methode

Dateien kopieren

Sie wollen eine oder mehrere Dateien aus einem Ordner in einen anderen kopieren, zum Beispiel, um eine Sicherheitskopie anzulegen.

Lösung

Mit dem Cmdlet *Copy-Item* (kurz: *copy*) kopieren Sie einzelne Dateien:

```
PS > Copy-Item $env:windir\windowsupdate.log $env:temp\update.log
PS > . $env:temp\update.log
```

Möchten Sie nur bestimmte Dateien aus einem Ordner an einen neuen Ort kopieren, verwenden Sie Platzhalterzeichen. Der Zielordner muss existieren, bevor Sie *Copy-Item* aufrufen:

```
PS > md $env:userprofile\Logfiles
PS > Copy-Item $env:windir\*.log -Destination $env:userprofile\Logfiles
PS > Get-ChildItem $env:userprofile\Logfiles
(...)
PS > (Get-ChildItem $env:userprofile\Logfiles).Count
73
```

Komplexere Kopieraufträge führen Sie am besten nach wie vor mit bewährten Befehlszeilentools wie *robocopy* durch:

```
PS > robocopy $env:windir $env:userprofile\Logfiles *.log /S /R:0 /NDL
```

HINWEIS

Das Tool *robocopy* ist seit Windows Vista und Server 2008 fester Bestandteil des Betriebssystems und kann bei älteren Windows-Versionen als Teil des kostenlosen Windows Resource Kit heruntergeladen werden.

Die obige von *robocopy* verrichtete Aufgabe lässt auch mit *Copy-Item* durchführen:

```
PS > md $env:userprofile\Logfiles
PS > Get-ChildItem $env:windir -filter *.log -recurse -ea SilentlyContinue | →
    Copy-Item -destination $env:userprofile\Logfiles
```

Allerdings gibt es dabei wesentliche Unterschiede. Weil *Copy-Item* sich ausschließlich auf das Kopieren von Daten beschränkt, kann es nicht mit NTFS-Berechtigungen umgehen und führt außerdem eine »flat copy« durch und kopiert sämtliche Dateien in einen einzelnen Ordner. Werkzeuge wie *robocopy* sind vielseitiger und verfügen über erheblich mehr praxistaugliche Optionen, um Kopierarbeiten zuverlässig durchzuführen. *Copy-Item* sollte daher nur verwendet werden, um einzelne wenige Dateien zu kopieren.

HINWEIS

Mit dem Cmdlet *Move-Item* verschieben Sie Dateien.

Hintergrund

Copy-Item prüft nicht, ob die Zielfeile bereits existiert, und überschreibt schon vorhandene Dateien.

```
PS > Copy-Item c:\autoexec.bat $env:temp\autoexec.bat.bak
PS > Copy-Item c:\autoexec.bat $env:temp\autoexec.bat.bak
PS > Copy-Item c:\autoexec.bat $env:temp\autoexec.bat.bak
```

Möchte man dies verhindern, muss man selbst mit *Test-Path* prüfen, ob die Zielfeile schon existiert:

```
PS > if (!(Test-Path $env:temp\autoexec.bat.bak)) {
    Copy-Item c:\autoexec.bat $env:temp\autoexec.bat.bak }
```

Externe Befehle wie *Robocopy* lassen sich in PowerShell-Code einbetten. Im folgenden Beispiel wird ein Filter verwendet, um die Ausgaben von *Robocopy* in Echtzeit zu verarbeiten und Fehlermeldungen in rot hervorzuheben. Fehler werden vom Filter durch das Wort »FEHLER« in Großbuchstaben identifiziert. Setzen Sie eine nicht-deutsche Version von *Robocopy* ein, passen Sie das Suchwort entsprechend an und ersetzen die deutschen Stichworte durch die Stichworte, die Sie jeweils farblich hervorheben wollen.

```
Filter Show-Errors {
    if ($_ -clike "*FEHLER*") {
        Write-Host -foregroundColor White -backgroundColor Red $_
        $marknext = $TRUE
    } elseif ($marknext) {
        $marknext = $FALSE
        Write-Host $_
    } else {
        $_
    }
}

PS > robocopy $env:windir $env:userprofile\Logfiles *.log /S /R:0 /NDL /NP | →
    Show-Errors
```

Ordner kopieren

Sie wollen einen Ordner einschließlich sämtlicher Unterordner kopieren.

Lösung

Verwenden Sie *Copy-Item* mit dem Parameter *-recurse*:

```
PS > Copy-Item $env:windir\web\wallpaper $home -recurse
PS > explorer.exe $env:temp\wallpaper
```

Robuster funktionieren bewährte Befehlszeilentools wie *xcopy*:

```
PS > xcopy $env:windir\web\wallpaper $env:temp\neuerordner\
(...)
PS > explorer.exe $env:temp\neuerordner
```

Hintergrund

Geben Sie *Copy-Item* als Quelle einen Ordner an, kopiert es diesen Ordner, aber nicht seinen Inhalt. Damit auch der Inhalt des Ordners kopiert wird, muss der Parameter *-recurse* angegeben werden. Der kopierte Ordner trägt anschließend denselben Namen wie der Quellordner.

Mehr Möglichkeiten bieten bewährte Befehlszeilentools wie *xcopy* oder *robocopy*. Mit *xcopy* können Sie dem Zielordner einen anderen Namen geben. Achten Sie darauf, an den Zielpfad ein abschließendes »\« anzuhängen, damit *xcopy* erkennen kann, dass das Ziel ein Ordner sein soll. Ohne Angabe von »\« fragt *xcopy* andernfalls nach, ob das Ziel ein Ordner oder eine Datei sein soll. Wählen Sie die Option *Datei*, werden alle Dateien in einer neuen Datei zusammengefasst, was selten sinnvoll ist.

Neue Ordner anlegen

Sie wollen einen oder mehrere neue Ordner anlegen.

Lösung

Legen Sie neue Ordner mit der Funktion *md* (*New-Item -type Directory*) an:

```
PS > md $env:temp\Neuer Ordner
```

Oder verwenden Sie *New-Item*:

```
PS > New-Item -Path c:\newfolder -ItemType Directory
```

Wurde der Ordner erfolgreich angelegt, liefern beide Befehle ein Ordnerobjekt zurück. Um diese Ausgabe zu verstecken, falls sie nicht erwünscht ist, leiten Sie alle Ausgaben weiter an *Out-Null*:

```
PS > New-Item -Path c:\newfolder -ItemType Directory | Out-Null
```

Hintergrund

Intern ruft die Funktion *md* das Cmdlet *New-Item* mit dem Parameter *-type Directory* auf. Sie können Ordner also auch selbst mit *New-Item* anlegen.

Enthält der angegebene Ordnername Leerzeichen, muss er in Anführungszeichen gestellt werden. Geben Sie einen relativen Pfadnamen an, wird er automatisch aufgelöst.

Geben Sie zum Beispiel nur einen Ordnernamen ohne Pfadangabe an, wird der neue Ordner im aktuellen Ordner angelegt. Dies funktioniert natürlich nur, wenn der aktuelle Ordner auf einen Ort im Dateisystem eingestellt ist. Haben Sie den aktuellen Ordner zum Beispiel auf einen anderen Bereich wie die Registrierungsdatenbank eingestellt, legt *md* keinen Ordner an, sondern einen Registrierungsschlüssel.

```
PS > New-Item HKCU:\Software\NewKey -ItemType Directory
```

```
Hive: HKEY_CURRENT_USER\Software
```

SKC	VC	Name	Property
---	---	----	-----
0	0	NewKey	{}

Existiert der übergeordnete Ordner Ihrer Pfadangabe noch nicht, wird dieser abhängig vom Provider möglicherweise ebenfalls angelegt. Im Dateisystem kann man auf diese Weise mit einer Zeile eine ganze Kette verschachtelter Ordner anlegen (in der Registrierungsdatenbank funktioniert dies nicht):

```
PS > md 'c:\test\Unterordner\Weiterer Ordner\Noch einer'
```

Existiert der Zielordner bereits, wird kein neuer Ordner angelegt und eine Fehlermeldung ausgegeben. Um diese zu vermeiden, könnten Sie zuerst mit *Test-Path* prüfen, ob der Zielordner bereits existiert.

```
PS > if ((Test-Path c:\neuerordner) -eq $false) { md c:\neuerordner | Out-Null }
```

Alternativ könnten Sie den Ordner auch einfach anzulegen versuchen und Fehlermeldungen ignorieren (trial & error). Dann allerdings würden Sie auch keine Fehlermeldung für andere Fehlerarten erhalten wie beispielsweise mangelnde Berechtigungen:

```
PS > md c:\neuerordner -ea SilentlyContinue | Out-Null
```

Neue Datei anlegen

Sie möchten eine neue Datei anlegen, zum Beispiel, um darin Informationen zu speichern.

Lösung

Legen Sie die neue Datei mit *New-Item* an und geben Sie als Typ *File* an:

```
PS > New-Item $env:temp\logbuch.txt -type File -value "Mein Text"
```

Der Ordner, in dem die Datei liegen soll, muss bereits existieren. Geben Sie den Parameter *-force* an, werden alle erforderlichen Ordner ebenfalls angelegt. In diesem Fall würde *New-Item* die Datei allerdings nun auch überschreiben, falls sie schon existiert. Diese Zeile legt ein Profilskript sowie alle dafür erforderlichen Ordner an, falls die Datei noch nicht existiert:

```
PS > If ((Test-Path $profile) -eq $false) →  
    { New-Item $profile -type File -force | Out-Null }
```

Sie können Informationen auch per Umleitung in eine Datei schreiben. Die Datei wird dabei automatisch für Sie angelegt. Existiert die angegebene Datei, wird sie überschrieben.

```
PS > "Mein Text" > $env:temp\logbuch.txt
```

Möchten Sie Informationen an eine vorhandene Datei anhängen, verwenden Sie entweder den Doppelumleitungspfeil oder das Cmdlet *Out-File* mit seinem Parameter *-append*:

```
PS > "Mein Text" >> $env:temp\logbuch.txt  
PS > "Mein Text" | Out-File -path $env:temp\logbuch.txt -append
```

Schließlich kann auch *Set-Content* neue Dateien anlegen:

```
PS > Set-Content logbuch.txt -value "Mein Text"
```

Hintergrund

Obwohl es viele Methoden gibt, um Informationen in einer Datei zu speichern, unterscheiden sie sich in feinen Details, die die Art und den Umfang der möglichen Optionen betreffen sowie das Encoding, also die Speicherform der Textinformationen.

Out-File ist dafür gedacht, Informationen, die normalerweise in die Konsole ausgegeben werden, stattdessen in eine Datei zu speichern. *Out-File* wandelt die Daten in der PowerShell-Pipeline dabei genauso um, wie es bei der Ausgabe in die Konsole auch geschieht. Deshalb ist das Ergebnis in der Datei identisch:

```
PS > Get-ChildItem | Out-File $env:temp\ausgabe.txt  
PS > Invoke-Item $env:temp\ausgabe.txt
```

TIPP

Leiten Sie Befehlsergebnisse auf diese Weise in eine Datei um, entspricht das Ergebnis exakt der Konsolenausgabe. Das kann unerwünscht sein, denn die Konsole hat eine relativ schmale Breite und muss daher häufig Informationen abschneiden. Für eine ungekürzte Fassung hängen Sie deshalb an den Befehl eine Kombination aus *Format-Table* und *Out-File* an. Sie sorgt dafür, dass die Breite der Textdatei größer ist als die der Konsole:

```
PS > Get-Process | Format-Table -AutoSize | →  
    Out-File $env:temp\report.txt -Width 10000  
PS > Invoke-Item $env:temp\report.txt
```

Abgekürzt werden jetzt nur noch Zeilen, die breiter sind als 10.000 Zeichen oder die mehrzeilige Inhalte aufweisen.

Set-Content schreibt die Eingabe ohne PowerShell-spezifische Umwandlung in die Datei. Weil es also PowerShell-Objekte nicht ausdrücklich in Text verwandelt, werden solche Objekte hier nur mit ihrer Standardeigenschaft dargestellt, was in der Regel der Typname ist. So gehen die Informationen in den Objekten verloren, weswegen sich *Set-Content* nur dafür eignet, eigene Texte in eine Datei zu schreiben:

```
PS > Get-ChildItem | Set-Content $env:temp\ausgabe.txt  
PS > Invoke-Item $env:temp\ausgabe.txt
```

Diese Unterschiede sind nur dann wichtig, wenn Sie PowerShell-Objekte in die Datei schreiben. Haben Sie PowerShell-Objekte vorher selbst in Text umgewandelt, verhalten sich beide Cmdlets gleich:

```
PS > Get-ChildItem | Out-String -Stream | Set-Content $env:temp\ausgabe.txt  
PS > Invoke-Item $env:temp\ausgabe.txt  
PS > Get-ChildItem | Out-File ausgabe.txt  
PS > Invoke-Item $env:temp\ausgabe.txt
```

Beide Cmdlets unterstützen den Parameter *-encoding*, über den Sie festlegen, in welchem Zeichensatz die Texte in die Datei geschrieben werden. Die folgenden Begriffe sind für den Parameter *-encoding* erlaubt:

```
PS > [System.Enum]::GetNames([Microsoft.PowerShell.Commands →  
    .FileSystemCmdletProviderEncoding])  
Unknown  
String  
Unicode  
Byte  
BigEndianUnicode  
UTF8  
UTF7  
Ascii
```

New-Item kann auch völlig leere Dateien anlegen und überschreibt Dateien normalerweise nicht (es sei denn, Sie geben den Parameter *-Force* an). Der Parameter *-Force* sorgt dafür, dass nicht nur die Datei angelegt wird, sondern zusätzlich alle Ordner, die im Dateipfad genannt werden und noch nicht existieren. Allerdings überschreibt *New-Item* jetzt ohne weitere Rückfrage die Datei, falls sie schon existiert.

Die folgende Zeile demonstriert, wofür dieses Verhalten nützlich sein kann. Die Zeile legt das Standard-PowerShell-Profilskript (einschließlich aller dafür nötigen Ordner) an, falls die Datei noch nicht existiert. Danach wird das Profilskript im Editor geöffnet. So können Sie jederzeit Ihrem Profilskript weitere Anweisungen hinzufügen:

```
PS> if (!(Test-Path $profile)) { New-Item $profile -type file -force | →  
    Out-Null}; Invoke-Item $profile
```

Text in eine Datei schreiben

Sie möchten Text in eine Datei schreiben.

Lösung

Möchten Sie, dass der Text eventuell vorhandenen Text in der Datei ersetzt, verwenden Sie entweder die Umleitung der Konsole:

```
PS > "Dies ist Text" > datei.txt
```

Oder Sie setzen *Set-Content* ein. Dieses Cmdlet erlaubt Ihnen zusätzlich, das Textencoding festzulegen:

```
PS > Set-Content datei.txt "Dies ist Text" -encoding Unicode
```

Wollen Sie Text an bereits vorhandenen Text einer Datei anhängen, verwenden Sie entweder das doppelte Umleitungszeichen:

```
PS > "Noch mehr Text" >> datei.txt
```

Oder Sie greifen zum Cmdlet *Add-Content*:

```
PS > Add-Content datei.txt "Dies ist Text" -encoding Unicode
```

Hintergrund

Text kann in unterschiedlicher Formatierung in einer Datei gespeichert werden (Encoding). Im Unicode-Format werden zum Beispiel pro Zeichen zwei Bytes verwendet, wodurch sich ein grö-

ßerer Zeichensatz ergibt. Verwenden Sie die Umleitung der Konsole, haben Sie keinen Einfluss auf das Encoding. Stattdessen verwendet die Umleitung das Encoding der Konsole.

Mit dem Cmdlets *Set-Content* und *Add-Content* gewinnen Sie Flexibilität hinzu, denn hier dürfen Sie mit dem Parameter *-encoding* das verwendete Encoding selbst auswählen. Zur Verfügung stehen die Werte aus *Microsoft.PowerShell.Commands.FileSystemCmdletProviderEncoding*:

```
PS > [System.Enum]::GetNames([Microsoft.PowerShell.Commands.FileSystemCmdletProviderEncoding])
Unknown
String
Unicode
Byte
BigEndianUnicode
UTF8
UTF7
Ascii
```

Das Encoding wird besonders dann wichtig, wenn Sie nicht Text, sondern Binärdaten in einer Datei speichern wollen. Die folgende Zeile würde das Standardencoding der Konsole verwenden und die angegebenen Zahlen als Text speichern:

```
PS > [byte[]]@(1,2,3,4,5) > test.txt
PS > .\test.txt
```

Wählen Sie dagegen das Encoding »Byte« zusammen mit *Set-Content*, werden die Zahlen als Bytearray binär in die Datei geschrieben:

```
PS > [byte[]]@(1,2,3,4,5) | Set-Content test.txt -encoding Byte
PS > .\test.txt
```

Eine besondere Rolle spielt *Out-File*. Auch mit diesem Cmdlet lassen sich Texte in eine Datei schreiben:

```
PS > "Eine Zeile" | Out-File test.txt
PS > "Noch eine Zeile" | Out-File test.txt -append
```

Out-File verhält sich auf den ersten Blick ähnlich wie *Set-Content* und *Add-Content*, kombiniert aber beide, denn mit dem Parameter *-append* bestimmen Sie, ob Informationen an eine vorhandene Datei angehängt oder vorhandene Informationen überschrieben werden sollen. Vor allem aber konvertiert *Out-File* etwaige PowerShell-Objekte zuerst in Text. Deshalb unterscheidet sich das Ergebnis der folgenden beiden Zeilen stark:

```
PS > Get-ChildItem | Set-Content test.txt
PS > .\test.txt
PS > Get-ChildItem | Out-File test.txt
PS > .\test.txt
```

Verwenden Sie *Out-File* immer dann, wenn Sie das Ergebnis eines Befehls in eine Textdatei umleiten und dabei sicherstellen wollen, dass es genauso repräsentiert wird wie in der Konsole. Alternativ könnten Sie das Ergebnis eines Befehls auch manuell zuerst mit *Out-String* in Text umwandeln:

```
PS > Get-ChildItem | Out-String | Set-Content test.txt
PS > .\test.txt
```

Text aus einer Datei lesen

Sie möchten den Inhalt einer Textdatei lesen und weiterverarbeiten.

Lösung

Verwenden Sie *Get-Content*, um den Inhalt der Datei zu lesen.

```
PS > $inhalt = $env:windir\windowsupdate.log
```

Sie erhalten den Inhalt der Textdatei zeilenweise als String-Array zurück. Deshalb können Sie die gelesenen Zeilen auch einzeln mit *Select-String* auswerten, um nur die Zeilen zu lesen, die ein bestimmtes Stichwort enthalten. Der folgende Code liest alle Textzeilen der Logbuchdatei, die das Schlüsselwort »Successfully Installed« enthalten. Durch das Encoding UTF8 bleiben deutsche Umlaute erhalten:

```
PS > Get-Content $env:windir\windowsupdate.log -Encoding UTF8 | Where-Object {
    $_ -like '*successfully installed*' }
```

Das Trennzeichen für das zeilenweise Lesen (normalerweise der Zeilenumbruch) kann mit dem Parameter *-delimiter* selbst gewählt werden. Eine tabulatorseparierte Textliste könnte also auch feldweise ausgelesen werden, wenn Sie als Trennzeichen ein Tabulatorzeichen angeben:

```
PS > Get-Content $env:windir\windowsupdate.log -delimiter `t -Encoding UTF8 |
    Where-Object { $_ -like '*successfully installed*' }
```

HINWEIS

Wenn Sie das Ergebnis begutachten, werden Sie ein Problem feststellen: Weil *Get-Content* nun nicht mehr nach Zeilenumbrüchen trennt, sondern nach Tabulatoren, verschmelzen jeweils das letzte Element einer Zeile und das nächste Element der folgenden Zeile.

Hintergrund

Get-Content erleichtert den Umgang mit textbasierten Logbuchdateien. So könnten Sie beispielsweise Informationen aus einer längeren Logbuchdatei extrahieren und in einer neuen Datei speichern:


```
PS > Get-Content $env:windir\windowsupdate.log -delimiter `t -Encoding UTF8 | →
    Where-Object { $_ -like '*successfully installed*' } | →
    Set-Content $env:temp\report.txt
PS > Invoke-Item $env:temp\report.txt
```

Get-Content gibt normalerweise gelesene Informationen sofort als *String* weiter an die Pipeline. Es kann besonders bei großen Dateien effizienter sein, die Informationen paketweise zu verarbeiten. Mit dem Parameter *-ReadCount* weisen Sie *Get-Content* an, die gelesenen Informationen paketweise zu verpacken und als Paket (Feld) an die Pipeline weiterzugeben. Der Parameter *-ReadCount 0* liefert dabei sämtliche Informationen in einem einzelnen Paket. Der Parameter *-ReadCount 1* entspricht der Vorgabe und verpackt die Informationen nicht. Jeder andere Wert schnürt Pakete mit der angegebenen Größe.

Wenn Sie das Ergebnis von *Get-Content* in der Pipeline weiterbearbeiten möchten, dürfen Sie den Parameter *-ReadCount* nicht ändern, weil die Folgebefehle in der Pipeline sonst mehrere Zeilen auf einmal zu bearbeiten hätten. Mit einer *foreach*-Schleife dagegen ist dies möglich. Sie sehen hier zwei unterschiedliche Ansätze, die beide dieselbe Aufgabe erledigen. Der erste Ansatz verwendet die PowerShell-Pipeline:

```
PS > Get-Content $env:windir\windowsupdate.log -Encoding UTF8 | →
    Where-Object { $_ -like '*successfully installed*' } | →
    ForEach-Object {($_ -split 'update: ')[-1]} | →
    Set-Content $env:temp\report.txt
PS > Invoke-Item $env:temp\report.txt
```

Der zweite Ansatz verzichtet auf die PowerShell-Pipeline und optimiert den Lesevorgang mit *-ReadCount*, sodass die gesamte Datei in einem Schritt gelesen wird:

```
PS > $zeilen = Get-Content $env:windir\windowsupdate.log -Encoding UTF8 -ReadCount 0
PS > $updates = foreach($zeile in $zeilen) { if ($zeile -like →
    '*successfully installed*' ) { ($zeile -split 'update: ')[-1]} }
PS > Set-Content $env:temp\report.txt -value $updates
PS > Invoke-Item $env:temp\report.txt
```

Das Ergebnis ist jeweils dasselbe, nämlich ein übersichtlicher Report der installierten Windows-Updates, aber der zweite Ansatz löst die Aufgabe rund fünfmal schneller. Dafür benötigt er mehr Speicherplatz, denn die gesamte Datei muss dafür zunächst in den Speicher geladen werden.

Inhalt einer Textdatei löschen

Sie möchten den Inhalt einer Textdatei komplett löschen, ohne jedoch die Datei selbst zu löschen.

Lösung

Löschen Sie den Dateiinhalt mit *Clear-Content*:

```
PS > Clear-Content test.txt
```

Hintergrund

Clear-Content entfernt den Inhalt einer Datei, ohne jedoch die Datei selbst zu löschen. Dabei unterstützt *Clear-Content* Platzhalterzeichen, sodass Sie auch mehrere Dateiinhalte in einem Schritt entfernen können:

```
PS > Clear-Content *.txt -WhatIf
```

Clear-Content kann den Inhalt schreibgeschützter Dateien nur löschen, wenn der Parameter *-Force* angegeben wurde:

```
PS > Set-Content test.txt "Eine Textzeile"
PS > attrib +r test.txt
PS > Clear-Content test.txt
Clear-Content : Der Zugriff auf den Pfad C:\Users\Tobias\test.txt wurde verweigert.
Bei Zeile:1 Zeichen:14
+ Clear-Content <<<< test.txt
PS > Clear-Content test.txt -force
```

Datei löschen

Sie möchten eine Datei löschen.

Lösung

Verwenden Sie *Del* (*Remove-Item*), um die Datei zu löschen:

```
PS > del c:\test.txt
```

Remove-Item unterstützt Platzhalterzeichen, sodass Sie auch mehrere Dateien auf einmal löschen können:

```
PS > del $home\*.txt -whatIf
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel
"C:\Users\Tobias\ausgabe.txt".
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel "C:\Users\Tobias\datei.txt".
(...)
PS > del $home\*.txt -confirm
```

Bestätigung

Möchten Sie diese Aktion wirklich ausführen?

Ausführen des Vorgangs "Datei entfernen" für das Ziel "C:\Users\Tobias\ausgabe.txt".

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe (Der Standardwert ist "J"):

Hintergrund

Remove-Item entfernt die angegebene Datei ohne weitere Rückfragen. Dieses Cmdlet kann über Platzhalterzeichen auch mehrere infrage kommende Dateien löschen. Damit Sie in diesem Fall nicht versehentlich zu viele Dateien löschen, sollten Sie Platzhalterzeichen sorgsam einsetzen und mit den Parametern *-whatIf* oder *-confirm* kombinieren.

Ist das Schreibschutzattribut einer Datei gesetzt, kann *Remove-Item* diese Datei nicht entfernen:

```
PS > Set-Content testdatei.txt "Eine Testdatei"
PS > attrib +r testdatei.txt
PS > Remove-Item testdatei.txt
Remove-Item : Das Element C:\Users\Tobias\testdatei.txt kann nicht entfernt werden:
Für das Ausführen des Vorgangs sind keine ausreichenden Berechtigungen vorhanden.
Bei Zeile:1 Zeichen:12
+ Remove-Item <<<< testdatei.txt
```

Verwenden Sie den Parameter *-force*, um sich über den Schreibschutz hinwegzusetzen:

```
PS > Remove-Item testdatei.txt -force
```

Möchten Sie mehrere Dateien löschen und die Dateien nicht anhand ihres Dateinamens auswählen, sondern aufgrund anderer Dateieigenschaften, verwenden Sie *dir* (*Get-ChildItem*) und einen Filter (*Where-Object*), um die gewünschten Dateien auszuwählen. Senden Sie das Ergebnis dann an *Remove-Item*. Die folgende Zeile löscht beispielsweise alle Textdateien in Ihrem Basisordner, die älter sind als 30 Tage:

```
PS > Get-ChildItem $home\*.txt | Where-Object { $_.CreationTime -lt →
(Get-Date).AddDays(-30) } | Remove-Item -whatIf
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel
"C:\Users\Tobias\ausgabe.txt".
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel
"C:\Users\Tobias\logbuch.txt".
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel "C:\Users\Tobias\test1.txt".
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel
"C:\Users\Tobias\testdaten.txt".
WhatIf: Ausführen des Vorgangs "Datei entfernen" für das Ziel
"C:\Users\Tobias\trans1.txt".
```

Remove-Item kann auch Ordner löschen. Ist der Ordner nicht leer, erscheint eine Sicherheitsabfrage, falls Sie nicht ausdrücklich den Parameter *-recurse* verwendet haben.

Datei oder Ordner umbenennen

Sie möchten eine Datei oder einen Ordner umbenennen.

Lösung

Benennen Sie die Datei mit *Rename-Item* um:

```
PS > "Test" > test.txt
PS > Rename-Item test.txt experiment.txt
PS > .\experiment.txt
```

Ordner lassen sich auf diese Weise ebenfalls umbenennen:

```
PS > md NeuerOrdner
PS > Rename-Item NeuerOrdner OrdnerNeu
PS > Get-ChildItem Ord*
```

Sie dürfen für den alten Namen des Objekts auch einen absoluten Pfadnamen verwenden, nicht jedoch für den neuen Namen des Objekts:

```
PS > md $env:temp\NewFolder
PS > Rename-Item $env:temp\NewFolder NeuerName
```

Hintergrund

Rename-Item verlangt von Ihnen zwei Angaben: den Namen des Objekts (Items), den Sie umbenennen wollen, sowie den neuen Namen. Während es Ihnen bei der ersten Angabe freigestellt ist, ob Sie einen relativen oder absoluten Pfadnamen angeben, handelt es sich bei der zweiten Angabe um den neuen Namen des Objekts, der also nicht als Pfadname formuliert sein darf.

ACHTUNG *Rename-Item* kann nur funktionieren, wenn es die Zielfeile noch nicht gibt. Sorgen Sie also vor dem Umbenennen dafür, dass nicht bereits eine Datei oder ein Ordner unter dem neuen Namen existiert.

Zwar unterstützt *Rename-Item* keine Platzhalterzeichen, aber mithilfe der PowerShell-Pipeline können Sie auch massenweise Dateien umbenennen. Das allerdings kann natürlich – wie bei allen automatisierten Prozessen – gefährlich sein, falls sich Fehler einschleichen. Setzen Sie deshalb in solchen Fällen die Parameter *-whatIf* oder *-confirm* ein.

Der folgende Code würde alle Textdateien in einem Ordner in Dateien mit der Dateierweiterung *.log* umbenennen:

```
PS > Get-ChildItem *.txt | Rename-Item -newname {
    { $_.Name -replace '.txt$', '.log' } -whatIf
```

Hierbei erhält *Rename-Item* den Namen des umzubennenden Objekts direkt über die Pipeline vom vorangegangenen *Get-ChildItem*-Befehl. Anstatt eines festen neuen Namens übergibt der Parameter *-newname* in diesem Fall einen ausführbaren Skriptblock, der für jede einzelne Datei ausgewertet wird. Der tatsächliche neue Name der Datei besteht dabei aus dem alten Dateinamen (zu finden in *\$_Name*), bei dem mit dem *-replace*-Operator die Endung *.txt* durch die

neue Endung *.log* ersetzt wird. Weil es sich beim Suchmuster um einen regulären Ausdruck handelt, sorgt das »\$« am Ende des Suchworts dafür, dass der Suchtext nur am Ende des Dateinamens gefunden wird, nicht aber, falls ein Dateiname an anderer Stelle den Text *.txt* enthalten sollte. Der Parameter *-whatIf* sorgt dafür, dass die Umbenennungen nur angezeigt, aber nicht wirklich durchgeführt werden.

Ein anderes Beispiel benennt Bilddateien um und verwendet dabei einen fortlaufend durchnummerierten Präfix.

```
PS > $bilder = [Environment]::GetFolderPath("MyPictures")
PS > Get-ChildItem $bilder *.jpg | ForEach-Object { $c=0} {
>> $newname = "Präfix{0:d4}.jpg" -f $c; Rename-Item $_.FullName →
    $newname -whatIf; $c+=1 }
>>
```

Sie könnten diesen Code weiter verfeinern, indem Sie die Liste der Bilddateien vor dem Umbenennen zuerst mit *Sort-Object* nach Alter oder Größe sortieren. Achten Sie allerdings darauf, dass *Rename-Item* nur funktioniert, wenn es die Zieldatei noch nicht gibt. Wollen Sie den Code mehrfach auf denselben Ordner anwenden, müssen Sie jeweils ein anderes Präfixwort verwenden:

```
PS > $bilder = [Environment]::GetFolderPath("MyPictures")
PS > Get-ChildItem $bilder *.jpg | Sort-Object CreationTime | ForEach-Object →
    { $c=0} {
>> $newname = "Bild{0:d4}.jpg" -f $c; Rename-Item $_.FullName $newname →
    -WhatIf; $c+=1 }
```

Wollen Sie lieber den Zeitstempel einer Datei zu ihrem neuen Namen machen, gehen Sie so vor:

```
PS > $bilder = [Environment]::GetFolderPath("MyPictures")
PS > Get-ChildItem $bilder *.jpg | ForEach-Object {
>> $newname = "{0:dd.MM.yy HH-mm-ss} Bild.jpg" -f $_.CreationTime; →
    Rename-Item $_.FullName $newname -whatIf}
```

Auf erweiterte Dateieigenschaften zugreifen

Sie möchten die erweiterten Dateieigenschaften lesen, die normalerweise im Eigenschaften-Diologfeld der Datei auf der Registerkarte *Details* zu sehen sind.

Lösung

An viele Dateien sind zusätzliche Informationen angeheftet, die sogenannten Metadaten. PowerShell zeigt diese Daten normalerweise nicht an. Um sie zu lesen, verwenden Sie das COM-Objekt *Shell.Application*.

```

PS > $bilder = [Environment]::GetFolderPath("MyPictures")
PS > $liste = Get-ChildItem $bilder | ? { -not $_.PSIsContainer }
PS > $bild = $liste[0]
PS > $helper = New-Object -comObject Shell.Application
PS > $ordner = $helper.Namespace($bild.DirectoryName)
PS > $datei = $ordner.ParseName($bild.Name)
PS > $ordner.GetDetailsOf($datei, -1)
Typ: PNG-Bild
Abmessungen: 494 x 91_
Größe: 2,21 KB

```

Sie erhalten auf diese Weise eine Liste sämtlicher Zusatzeigenschaften der angegebenen Datei. Das lässt sich auch in einer Funktion kapseln:

```

Function Get-ExtendedProperties([String]$path) {
    $fileobject = Get-Item $path
    $helper = New-Object -comObject Shell.Application
    $ordner = $helper.Namespace($fileobject.DirectoryName)
    $datei = $ordner.ParseName($fileobject.Name)
    $ordner.GetDetailsOf($datei, -1)
}

PS > Get-ExtendedProperties $env:windir\explorer.exe
Dateibeschreibung: Windows-Explorer
Firma: Microsoft Corporation
Dateiversion: 6.0.6001.18000
Erstelldatum: _04._04._2008 __18:33
Größe: 2,79 MB

```

Dabei kann es passieren, dass Sonderzeichen ausgegeben werden, beispielsweise ein Copyright-Zeichen. Diese Zeichen werden in der Konsole nur dann richtig ausgegeben, wenn Sie einen TrueType-Zeichensatz verwenden. Verwenden Sie dagegen eine einfache Rasterschrift, stellt die Konsole die Zeichen als Fragezeichen dar. Die Schriftart der Konsole ändern Sie per Rechtsklick auf das Symbol der Konsole ganz links in der Titelleiste. Wählen Sie dann im Kontextmenü *Eigenschaften* und ändern Sie die Schriftart im Dialogfeld auf der Registerkarte *Schriftart*.

Hintergrund

Das COM-Objekt *Shell.Application* wird intern vom Windows-Explorer verwendet, um Ordner und Ordnerinhalte darzustellen. Über die Methode *Namespace()* navigieren Sie zu einem Ordner und der Ordner liefert mit seiner Methode *ParseName()* eine Datei aus dem Ordner.

Mit der Methode *GetDetailsOf()* des Ordners greifen Sie auf die erweiterten Eigenschaften einer Datei zu. Am einfachsten funktioniert dies, wenn Sie als Eigenschaftenkennziffer *-1* angeben, denn dann werden die wichtigsten definierten Eigenschaften korrekt beschriftet als Text ausgegeben. Dies war der Ansatz in den Beispielen oben.

Alternativ können Sie aber auch auf einzelne erweiterte Eigenschaften gezielt zugreifen und auf diese Weise vor allem sehr viel mehr Eigenschaften zutage fördern. Die Namen dieser Eigenschaften erfahren Sie über ihre Kennziffern, die je nach Windows-Version im Wertebereich 0 bis

287 liegen. Nicht alle Kennziffern sind definiert. Bei Windows Vista und erneut bei Windows 7 wurden weitere erweiterte Eigenschaften in die Liste aufgenommen.

```
PS > $helper = New-Object -comObject Shell.Application
PS > $ordner = $helper.Namespace($env:windir)
PS > 0..287 | ForEach-Object {'{0:000} = {1}' -f $_, →
    $ordner.GetDetailsOf($null, $_) }
000 = Name
001 = Größe
002 = Elementtyp
003 = Änderungsdatum
004 = Erstelldatum
005 = Letzter Zugriff
006 = Attribute
007 = Offlinestatus
008 = Offline verfügbar
(...)
```

Möchten Sie eine bestimmte Eigenschaft einer Datei auslesen, geben Sie die gewünschte Kennziffer an. Die Größe einer Datei verbirgt sich zum Beispiel hinter der Kennziffer 1:

```
PS > $datei = $ordner.ParseName("explorer.exe")
PS > $ordner.GetDetailsOf($datei, -1)
Dateibeschreibung: Windows-Explorer
Firma: Microsoft Corporation
Dateiversion: 6.0.6001.18000
Erselldatum: _04._04._2008 __18:33
Größe: 2,79 MB
PS > $ordner.GetDetailsOf($datei, 1)
2,79 MB
```

Sie könnten nun versuchen, sämtliche Eigenschaften einer Datei auf diese Weise auszugeben, würden aber schnell feststellen, dass viele davon nicht mit Werten belegt sind:

```
PS > 0..266 | % { "ID {0,3} = {1}" -f $_, $ordner.GetDetailsOf($datei, $_) }
ID 0 = explorer.exe
ID 1 = 2,79 MB
ID 2 = Anwendung
ID 3 = 18.01.2008 23:33
ID 4 = 04.04.2008 18:33
ID 5 = 04.04.2008 18:33
ID 6 = A
ID 7 =
ID 8 =
ID 9 = Anwendung
ID 10 = TrustedInstaller
ID 11 = Programm
ID 12 =
ID 13 =
ID 14 =
(...)
```

Dateiattribute lesen und ändern

Sie möchten ein oder mehrere Dateiattribute einer Datei ändern.

Lösung

Verwenden Sie den Konsolenbefehl *attrib.exe*. Die folgende Zeile setzt das Schreibschutzattribut der Datei *test.txt*:

```
PS > attrib +r test.txt
PS > attrib test.txt
A      R      C:\Users\Tobias\test.txt
```

Alternativ greifen Sie mit *Get-Item* auf die Datei zu und können nun wichtige Attribute direkt über Eigenschaften ändern:

```
PS > $datei = Get-Item test.txt
PS > $datei.IsReadOnly
True
PS > $datei.IsReadOnly = $true
PS > $datei.IsReadOnly
True
PS > $datei.IsReadOnly = $false
PS > $datei.IsReadOnly
False
PS > attrib test.txt
A      C:\Users\Tobias\test.txt
```

Hintergrund

Der einfachste Weg, Attribute einer Datei zu setzen oder zu löschen, ist der bewährte *attrib.exe*-Befehl, denn mit ihm brauchen Sie sich nicht selbst um binäre Additionen zu kümmern. Geben Sie einfach an, welches Attribut Sie setzen oder löschen möchten:

```
PS > attrib +h test.txt
PS > attrib -h test.txt
PS > attrib /?
Zeigt Dateiattribute an oder ändert sie.

ATTRIB [+R | -R] [+A | -A ] [+S | -S] [+H | -H] [+I | -I]
        [Laufwerk:] [Pfad] [Dateiname] [/S [/D] [/L]]

+   Setzt ein Attribut.
-   Löscht ein Attribut.
R   Attribut für 'Schreibgeschützte Datei'
A   Attribut für 'Zu archivierende Datei'
S   Attribut für 'Systemdatei'
H   Attribut für 'Versteckte Datei'
```



```

I   Attribut für 'Inhalt nicht indiziert'.
[Laufwerk:][Pfad][Dateiname]
    Gibt Dateien für den Attributprozess an.
/S  Verarbeitet passende Dateien im aktuellen Ordner
    und in allen Unterordnern.
/D  Verarbeitet auch die Ordner.
/L  Verarbeitet die Attribute des symbolischen Links anstelle
    des Linkziels

```

Attrib unterstützt Platzhalterzeichen und rekursive Aufrufe, sodass Sie in einer Zeile die Attribute vieler verschiedener Dateien setzen und löschen können.

```
PS > attrib *.txt -a /S
```

Möchten Sie die Dateiattribute lieber mit Bordmitteln setzen und löschen, müssen Sie sich einzelne Dateien mit *Get-Item* beschaffen oder das Ergebnis von Cmdlets wie *Get-ChildItem* auswerten, die Dateiobjekte zurückliefern.

Sie können danach die Dateiattribute auslesen oder komplett neu setzen:

```

PS > $file = Get-Item test.txt
PS > $file.Attributes
Archive
PS > $file.Attributes = "ReadOnly", "System"
PS > $file.Attributes
ReadOnly, System

```

Etwas schwieriger ist es, einzelne Attribute zu setzen oder zu löschen, ohne die übrigen Attribute zu verändern, denn dafür benötigen Sie binäre Operatoren:

```

PS > $file.Attributes
ReadOnly, System
PS > # ReadOnly Attribut entfernen
PS > $file.Attributes = $file.Attributes -band -not [IO.FileAttributes]"ReadOnly"
PS > $file.Attributes
Normal
PS > # ReadOnly Attribut hinzufügen
PS > $file.Attributes = $file.Attributes -bor [IO.FileAttributes]"ReadOnly"
PS > $file.Attributes
ReadOnly
PS > # ReadOnly- und Archiv-Attribut hinzufügen
PS > $file.Attributes = $file.Attributes -bor [IO.FileAttributes] →
    "ReadOnly,Archive"
PS > $file.Attributes
ReadOnly, Archive
PS > # Archiv- und System-Attribut löschen
PS > $file.Attributes = $file.Attributes -band -not [IO.FileAttributes] →
    "Archive,System"
PS > $file.Attributes
Normal

```

Dateiobjekte verfügen außerdem über die Eigenschaft *Mode*, die die gesetzten Attribute in einer Abkürzungsform anzeigt:

```
PS > $file = Get-Item test.txt
PS > $file.Mode
-----
PS > $file.Attributes = "ReadOnly", "System"
PS > $file.Mode
--r-s
```

Mode ist eigentlich eine Skripteigenschaft. Jedes Mal, wenn Sie Mode abfragen, wird intern der folgende PowerShell-Skriptcode ausgeführt:

```
PS > ($file | Get-Member mode).Definition
System.Object Mode {get=$catr = "";
    if ( $this.Attributes -band 16 ) { $catr += "d" } else { $catr += "-" } ;
    if ( $this.Attributes -band 32 ) { $catr += "a" } else { $catr += "-" } ;
    if ( $this.Attributes -band 1 ) { $catr += "r" } else { $catr += "-" } ;
    if ( $this.Attributes -band 2 ) { $catr += "h" } else { $catr += "-" } ;
    if ( $this.Attributes -band 4 ) { $catr += "s" } else { $catr += "-" } ;
    $catr;}
```

Dateien mit einem bestimmten Attribut finden

Sie möchten nur Dateien aus einem Ordner auflisten, bei denen ein bestimmtes Dateiattribut gesetzt ist.

Lösung

Schicken Sie das Ergebnis von *dir* (*Get-ChildItem*) durch einen Filter (*Where-Object*, kurz »?«) und überprüfen Sie für jede Datei, ob das gewünschte Attribut gesetzt ist.

```
PS > Get-ChildItem | Where-Object { $_.Attributes -band →
    [IO.FileAttributes]"ReadOnly" }
```

Hintergrund

Die Dateiattribute stehen in der Eigenschaft *Attributes* als Bitmaske zur Verfügung. Sie benötigen also einen binären Vergleich (Operator *-band*), um zu überprüfen, ob das gewünschte Attribut gesetzt ist oder nicht.

Damit die Attribut-Bitmasks leichter zu verstehen sind, gibt es für jedes Bit einen Klartextnamen in der Aufzählung *IO.FileAttributes*. Diese Aufzählung nennt Ihnen auch die Klartextnamen sämtlicher möglichen Attribute:

```
PS > [System.Enum]::GetNames([IO.FileAttributes])
ReadOnly
Hidden
System
Directory
Archive
Device
Normal
Temporary
SparseFile
ReparsePoint
Compressed
Offline
NotContentIndexed
Encrypted
```

Hinter diesen Namen verbergen sich einfache Zahlenwerte. Mit der folgenden Funktion listen Sie beliebige Aufzählungen sowie die zugrunde liegenden Werte auf:

```
Function Get-Enum{
    [Enum]::GetValues($args[0]) | Select-Object @{n="Name"; →
    e={$_.Name},@{n="Value";e={$_.Value}} |
    ft -auto
}
```

```
PS > Get-Enum IO.FileAttributes
```

Name	Value
ReadOnly	1
Hidden	2
System	4
Directory	16
Archive	32
Device	64
Normal	128
Temporary	256
SparseFile	512
ReparsePoint	1024
Compressed	2048
Offline	4096
NotContentIndexed	8192
Encrypted	16384

Wie sich herausstellt, wird das Attribut *ReadOnly* durch den Zahlenwert 1 repräsentiert, ist also das erste Bit in der Bitmaske. Sie könnten deshalb alle Dateien, bei denen dieses Attribut gesetzt ist, auch mit weniger Code ermitteln:

```
PS > # Die folgenden beiden Zeilen funktionieren identisch:
PS > Get-ChildItem | Where-Object { $_.Attributes -band →
    [IO.FileAttributes]"ReadOnly" }
PS > Get-ChildItem | Where-Object { $_.Attributes -band 1 }
```

Die erste (längere) Schreibweise ist allerdings besser lesbar.

Möchten Sie Dateien finden, bei denen mindestens eines von mehreren Attributen gesetzt ist, listen Sie die Attribute kommasepariert auf. Sie dürfen alternativ auch die Zahlenwerte der Attribute addieren. Die folgende Zeile findet alle Dateien, bei denen entweder das *ReadOnly*- oder das *Directory*-Attribut gesetzt ist:

```
PS > Get-ChildItem | Where-Object { $_.Attributes -band →
    [IO.FileAttributes]"ReadOnly,Directory" }
PS > Get-ChildItem | Where-Object { $_.Attributes -band 17 }
```

Dies ist so, weil *-band* nun einen Wert ungleich null zurückliefert, sobald mindestens ein Attribut gesetzt ist. Alle Werte ungleich null werden in den booleschen Wert *\$true* übersetzt, der Vergleich trifft also zu.

Möchten Sie Dateien finden, bei denen sämtliche Attribute gesetzt sind, muss das Ergebnis des *-band*-Vergleichs genau dem Wert entsprechen, gegen den Sie verglichen haben:

```
PS > Get-ChildItem | Where-Object { $_.Attributes -band →
    [IO.FileAttributes]"ReadOnly,Directory" -eq →
    [IO.FileAttributes]"ReadOnly,Directory" }
PS > Get-ChildItem | Where-Object { $_.Attributes -band 17 -eq 17 }
```

Dateinamen mit speziellen Zeichen verarbeiten

Sie möchten mit Dateien arbeiten, die in ihrem Namen Sonderzeichen enthalten, die von PowerShell normalerweise mit einer besonderen Bedeutung versehen sind.

Lösung

Verwenden Sie den Parameter *-literalPath*, um den Dateinamen anzugeben.

```
PS > md [Test]
New-Item : Das Element mit dem angegebenen Namen C:\Users\Tobias\[Test] ist bereits
vorhanden.
Bei Zeile:1 Zeichen:34
+ param([string[]]$paths); New-Item <<<< -type directory -path $paths
PS > del [Test] -whatIf
PS > del -literalPath [Test] -whatIf
WhatIf: Ausführen des Vorgangs "Verzeichnis entfernen" für das Ziel
"C:\Users\Tobias\[Test]".
```

Hintergrund

PowerShell interpretiert einige Sonderzeichen als Platzhalterzeichen. Sind diese Platzhalterzeichen in Wirklichkeit Teil des Pfadnamens, werden sie von PowerShell falsch verstanden. Die folgende Anweisung löscht also zum Beispiel nicht den Ordner namens »[test]«, sondern in Wirklichkeit sämtliche Dateien und Ordner, die »t«, »e« oder »s« heißen:

```
PS > del [Test] -whatIf
```

Verwenden Sie dagegen den Parameter *-LiteralPath*, ignoriert PowerShell sämtliche Platzhalterzeichen und betrachtet sie als normale Pfadbestandteile:

```
PS > del -literalPath [Test] -whatIf
```

Sonderzeichen	Beschreibung
*	Ein oder mehrere beliebige Zeichen: <i>dir *.txt</i>
?	Genau ein beliebiges Zeichen: <i>dir *.?xt</i>
[abc]	Alle in den eckigen Klammern aufgeführten Zeichen: <i>dir *.t[axf]t</i>
[a-z]	Ein Zeichen im angegebenen Bereich: <i>dir *.t[f-z]t</i>

Tabelle 7.6 Besondere Platzhalterzeichen in Pfadangaben

Eine weitere Gefahr sind automatische Ersetzungen. Immer, wenn Sie einen Text in doppelte Anführungszeichen setzen, werden darin enthaltene Variablen und Sonderzeichen automatisch ersetzt. Möchten Sie zum Beispiel einen neuen Ordner anlegen, in dessen Name ein »\$«-Zeichen vorkommt, stellen Sie sicher, dass Sie den Namen nicht in doppelte Anführungszeichen stellen. Verwenden Sie entweder keine Anführungszeichen (wenn der Name keine Leerzeichen enthält) oder einfache Anführungszeichen:

```
PS > md "Spezial$Ordner"  
PS > md 'Spezial$Ordner'
```

Warum sich beide Anweisungen unterschiedlich verhalten, erkennen Sie am besten, wenn Sie sich den angegebenen Text direkt ausgeben. Dabei stellt sich heraus, dass PowerShell beim Text in doppelten Anführungszeichen »\$Ordner« als Variable interpretiert und durch ihren Wert ersetzt. Gibt es diese Variable gar nicht, wird ein Leerwert ersetzt und also ein Ordner namens »Spezial« angelegt.

```
PS > "Spezial$Ordner"  
Spezial  
PS > 'Spezial$Ordner'  
Spezial$Ordner
```

Ähnliches passiert, wenn Sie innerhalb doppelter Anführungszeichen den besonderen Backtick verwenden, der bei PowerShell wie ein Escape-Zeichen funktioniert. Die folgende Zeile würde den Ordner mit einem Tabulatorzeichen in der Mitte anlegen und weil Tabulatoren in Dateinamen nicht erlaubt sind, erhalten Sie eine Fehlermeldung:

```
PS > md "Spezial`tOrdner"
```

Um im Pfadnamen Backtick-Zeichen zu verwenden, setzen Sie wieder einfache Anführungszeichen ein:

```
PS > md 'Spezial`t0rdner'
```

Sie können all dies sogar kombinieren. Wollen Sie innerhalb des Pfadnamens bestimmte Teile automatisch ersetzen und andere nicht, markieren Sie die nicht zu ersetzenden »\$«-Zeichen mit einem Backtick:

```
PS > md "$env:username Daten ` $"
```

Aktuellen Ordner bestimmen (oder setzen)

Sie möchten feststellen, in welchem Ordner Sie sich augenblicklich befinden, oder den aktuellen Ordner neu festlegen.

Lösung

Verwenden Sie *Get-Location*, um den aktuellen Ordner zu bestimmen. Setzen Sie *Set-Location* (oder *cd*) ein, um den aktuellen Ordner neu festzulegen.

```
PS > Push-Location
PS > Get-Location
Path
----
C:\Users\Tobias

PS > cd $env:windir
PS > Get-Location
Path
----
C:\Windows

PS > Pop-Location
PS > Get-Location
Path
----
C:\Users\Tobias
```

Hintergrund

Mit *Get-Location* erfahren Sie den aktuellen Ordner, in dem Sie sich gerade befinden. Der aktuelle Ordner spielt eine große Rolle, wenn Sie relative Pfadnamen einsetzen, denn die werden immer ausgehend vom aktuellen Ordner bestimmt.

```
PS > Resolve-Path ..\*.*
PS > Resolve-Path \*
Path
----
C:\backup
C:\DRIVERS
C:\e807011a8f34765a18
(...)

PS > cd $env:windir
PS > Resolve-Path ..\*.*
Path
----
C:\autoexec.bat
C:\BOOTSECT.BAK
C:\config.sys
C:\Log.txt
C:\setup.log
C:\syslevel.lgl
C:\tvtpktfilter.dat
C:\wcid0.log

PS > Resolve-Path \*
Path
----
C:\backup
C:\DRIVERS
C:\e807011a8f34765a18
C:\Intel
C:\logbuchdateien
(...)
```

Mit *Set-Location* (oder kurz *cd*) legen Sie den aktuellen Ordner neu fest. Hierbei sind absolute und relative Pfadnamen erlaubt. Die folgende Anweisung wechselt zum Beispiel in das Wurzelverzeichnis (den obersten Ordner) des aktuellen Laufwerks:

```
PS > cd \
```

Bevor Sie den aktuellen Ordner wechseln, können Sie den aktuellen Ordner mit *Push-Location* speichern. Er wird dabei auf einen sogenannten Stack gelegt, von dem Sie ihn mit *Pop-Location* wiederherstellen. Dies funktioniert normalerweise nur genau in umgekehrter Reihenfolge, das heißt, *Pop-Location* stellt den Ordnerpfad wieder her, den Sie zuletzt mit *Push-Location* auf den Stack gelegt haben.

Mithilfe des Parameters *-stackname* sind Sie jedoch in der Lage, mehrere Stapel zu bilden, für die jeweils allerdings die Einschränkung erhalten bleibt, dass dort gespeicherte Ordner nur in umgekehrter Reihenfolge wieder abgerufen werden können:

```
PS > Push-Location -stackname Start
PS > cd $env:windir
PS > Push-Location -stackname Windows
PS > cd hklm:
PS > Pop-Location -stackname Start
PS > Get-Location
Path
----
C:\Windows

PS > Pop-Location -stackname Windows
PS > Get-Location
Path
----
C:\Windows
```

Sobald das letzte Element vom Stapel genommen wird, löscht PowerShell den Stapel wieder. Sie können ein darauf abgelegtes Element deshalb nur ein einziges Mal abrufen. Versuchen Sie es ein zweites Mal, erhalten Sie einen Fehler:

```
PS > Pop-Location -stackname Start
Pop-Location : Der Speicherstapel "Start" kann nicht gefunden werden. Er ist nicht
vorhanden oder ist kein Container.
Bei Zeile:1 Zeichen:13
+ Pop-Location <<<< -Stackname Start
PS > Get-Location
Path
----
C:\Windows
```

Pfadnamen auflösen

Sie möchten einen internen PowerShell-Pfadnamen auflösen und den »echten« Pfadnamen bestimmen.

Lösung

Verwenden Sie die interne PowerShell-Methode *GetUnresolvedProviderPathFromPSPath()*. Die folgende Zeile liefert Ihnen den echten Pfadnamen des aktuellen Ordners:

```
PS > $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPath →
((Get-Location))
C:\Users\Tobias
```

Ebenso gut können Sie jeden anderen Pfadnamen auflösen:


```
PS > $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPath →
('HKLM:')
HKEY_LOCAL_MACHINE\
```

Hintergrund

Get-Location liefert stets den Pfadnamen aus Sicht von PowerShell. Diese Sichtweise hat außerhalb von PowerShell allerdings nicht unbedingt Gültigkeit. Zum Beispiel werden virtuelle PowerShell-Laufwerke nur innerhalb von PowerShell anerkannt:

```
PS > New-PSDrive skripts filesystem $home
Name      Provider      Root                      CurrentLocation
----      -
skripts   FileSystem     C:\Users\Tobias
PS > cd skripts:
PS > Get-Location
Path
----
skripts:\
```

Um den PowerShell-internen Pfadnamen in einen allgemeingültigen Namen umzuwandeln, greifen Sie von PowerShell aus mit *Get-Item* auf den PowerShell-internen Pfad zu. Sie erhalten so ein Verzeichnisobjekt, das Ihnen in seiner Eigenschaft *FullName* den tatsächlichen allgemeinen Pfad meldet:

```
PS > (Get-Item (Get-Location)).FullName
C:\Users\Tobias\
```

Das funktioniert allerdings nur innerhalb des Dateisystems, denn nur dort steht die Eigenschaft *FullName* zur Verfügung. Registrierungsdatenbank-Pfade lösen Sie mit der Eigenschaft *Name* auf:

```
PS > cd HKLM:\Software\Microsoft
PS > Get-Location
Path
----
HKLM:\Software\Microsoft
PS > (Get-Item (Get-Location)).Name
HKEY_LOCAL_MACHINE\Software\Microsoft
```

Ein universellerer Ansatz macht sich eine interne PowerShell-Methode *GetUnresolvedProviderPathFrom-PSPath()* zunutze, die jeden beliebigen spezifischen PowerShell-Pfad in allgemeine Pfadangaben übersetzt:

```

PS > New-PSDrive skripts filesystem $home
Name      Provider      Root      CurrentLocation
----      -
skripts    FileSystem      C:\Users\Tobias

PS > cd skripts:
PS > Get-Location
Path
----
skripts:\

PS > $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPath →
    ((Get-Location))
C:\Users\Tobias\
PS > cd HKLM:\Software\Microsoft
PS > $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPath →
    ((Get-Location))
HKEY_LOCAL_MACHINE\Software\Microsoft

```

Veränderungen an Ordnerinhalten per Snapshot auswerten

Sie wollen den Ist-Zustand eines Ordners gegen einen früheren Zustand vergleichen, um beispielsweise herauszufinden, welche Änderungen eine Softwareinstallation vorgenommen hat. Oder Sie wollen den Inhalt zweier Ordner auf zwei unterschiedlichen Computern miteinander vergleichen, um herauszufinden, worin sich die Ordnerinhalte unterscheiden.

Lösung

Legen Sie sich einen Ausgangsschnappschuss an und vergleichen Sie den aktuellen Inhalt des Ordners anschließend mit *Compare-Object*. Die folgenden Zeilen überprüfen den Inhalt Ihres Benutzerprofils und finden alle Dateien, bei denen entweder Name oder Größe (Eigenschaften *Name* oder *Length*) sich geändert haben:

```

PS > $vorher = Get-ChildItem $home
PS > # Nehmen Sie Änderungen vor

PS > $nachher = Get-ChildItem $home

PS > Compare-Object $vorher $nachher -property Name, Length
PS > Compare-Object $vorher $nachher -PassThru -property Name, Length | →
    Where-Object { $_.SideIndicator -eq '=>' }

```

Möchten Sie Schnappschüsse über einen längeren Zeitraum oder zwischen unterschiedlichen Computern vergleichen, »serialisieren« Sie die Schnappschüsse mit *Export-CliXML* als XML-Datei. Anschließend können Sie diese Dateien auf Ihrem System mit *Import-CliXML* wieder als Schnappschuss einlesen und miteinander vergleichen.

Hintergrund

Compare-Object kann zwei Datensätze miteinander vergleichen und liefert die Änderungen zwischen beiden Listen. Dabei ist es wichtig, mit dem Parameter *-property* die Eigenschaften (Spaltennamen) anzugeben, nach denen Sie vergleichen möchten.

Compare-Object liefert die Unterschiede der beiden Datensätze zurück und zeigt in der Spalte *SideIndicator* an, ob diese im ersten oder im zweiten Datensatz liegen. Übersichtlicher wird die Ergebnisliste, wenn Sie sie nach dem Namen sortieren.

Geben Sie zusätzlich den Parameter *-passThru* an, werden die Originalobjekte wieder zurückgeliefert. Die Spalte *SideIndicator* ist nach wie vor darin vorhanden, jedoch normalerweise nicht sichtbar. Verwenden Sie *Select-Object*, um die Spalte explizit anzeigen zu lassen. Die folgende Zeile vergleicht den Status laufender Dienste und kann diesen Vergleich auch zwischen verschiedenen Computern anstellen. Dazu erstellen Sie den Schnappschuss jeweils als XML-Datei und fügen darin den Namen der untersuchten Maschine hinzu:

```
PS > Get-Service | ForEach-Object { $_ | Add-Member -memberType NoteProperty -Name ComputerName -Value $env:computername -passthru } | Export-CliXML $env:temp\dienstliste1.xml
```

Führen Sie diese Zeile also auf beiden Computern aus und speichern Sie das Ergebnis als *dienstliste1.xml* und *dienstliste2.xml*. Danach kopieren Sie die beiden XML-Dateien auf Ihr System und laden die Schnappschüsse:

```
PS > $snap1 = Import-CliXML $env:temp\dienstliste1.xml
PS > $snap2 = Import-CliXML $env:temp\dienstliste2.xml
```

Nun können beide Schnappschüsse miteinander verglichen werden. Da Sie Unterschiede in der Dienstkonfiguration erkennen wollen, sollen die beiden Eigenschaften *Name* und *Status* verglichen werden. Da beim Anlegen der Schnappschüsse eine neue Eigenschaft namens *ComputerName* hinzugefügt wurde, kann diese nun bequem angezeigt werden, um festzustellen, auf welchem der beiden Computer jeweils die Unterschiede vorliegen:

```
PS > Compare-Object $snap1 $snap2 -property Name, Status -passThru |
Select-Object Status, Name, Computername | Sort-Object Name
```

Status	Name	ComputerName
Running	HomeGroupListener	DEM05
Running	MMCSS	DEM05
Stopped	MMCSS	PC003
Stopped	Spooler	DEM05
Running	Spooler	PC003

Echtzeit-Überwachung von Änderungen an Ordnern

Sie möchten einen Ordner überwachen und alarmiert werden, wenn darin Änderungen vorgenommen werden, also zum Beispiel neue Dateien hinzugefügt, geändert oder gelöscht werden.

Lösung

Verwenden Sie *FileSystemWatcher* von .NET Framework. Der folgende Code überwacht Ihren Profilordner und meldet, wenn sich der Ordnerinhalt ändert:

```
PS > $fsw = New-Object System.IO.FileSystemWatcher
PS > $fsw.Path = $home
PS > $result = $fsw.WaitForChanged("All")
PS > $result
```

ChangeType	Name	OldName	TimedOut
-----	----	-----	-----
	Renamed Briefe	Korrespondenz	False

Der Aufruf von *WaitForChanged()* blockiert Ihr Skript. Es setzt seine Arbeit erst fort, wenn tatsächlich eine Änderung eingetreten ist. Möchten Sie ein Timeout festlegen, geben Sie dieses in Millisekunden mit an:

```
PS > $fsw = New-Object System.IO.FileSystemWatcher
PS > $fsw.Path = $home
PS > $result = $fsw.WaitForChanged("All", 5000)
PS > if ($result.TimedOut) {
>> "Keine Änderungen"
>> } else {
>> $result
>> }
>>
Keine Änderungen
```

Hintergrund

Das *FileSystemWatcher*-Objekt überwacht den angegebenen Ordner, sobald Sie *WaitForChanged()* aufrufen. Mit dieser Methode legen Sie fest, auf welche Änderungsereignisse Sie reagieren wollen. Geben Sie »All« an, liefert *WaitForChanged()* das erste beliebige Änderungsereignis. Andere mögliche Einstellungen liefern die folgende Aufzählung:

```
PS > [System.Enum]::GetNames([System.IO.WatcherChangeTypes])
Created
Deleted
Changed
Renamed
All
```

Einstellung	Beschreibung
<i>All</i>	Alle Änderungsereignisse
<i>Created</i>	Neue Dateien oder Unterordner wurden angelegt
<i>Deleted</i>	Vorhandene Dateien oder Unterordner wurden gelöscht
<i>Changed</i>	Vorhandene Dateien oder Unterordner wurden geändert
<i>Renamed</i>	Vorhandene Dateien oder Unterordner wurden umbenannt

Tabelle 7.7 Überwachungseinstellungen

Optional können Sie *WaitForChanged()* als zweitem Parameter einen Timeout in Millisekunden angeben. Tritt keines der überwachten Ereignisse innerhalb des Timeouts auf, beendet *WaitForChanged()* die Überwachung. Das Ergebnis der Überwachung ist ein Objekt mit einer Reihe von Eigenschaften, die Ihnen verraten, welche Änderung eingetreten ist oder ob das Timeout erreicht wurde.

Als Vorgabe überwacht *WaitForChanged()* nur den angegebenen Ordner, der existieren muss. Möchten Sie auch den Inhalt seiner Unterordner in die Überwachung einbeziehen, setzen Sie die Eigenschaft *includeSubdirectories* auf *\$true*:

```
PS > $fsw = New-Object System.IO.FileSystemWatcher
PS > $fsw.Path = $home
PS > $fsw.includeSubdirectories = $true
PS > $result = $fsw.WaitForChanged("All")
PS > $result
```

Da *WaitForChanged()* ein synchroner Aufruf ist, der Ihr Skript blockiert, erhält der Anwender während der Überwachung keine Rückmeldung. Sie könnten eine Rückmeldung aber erzeugen, indem Sie die Überwachung regelmäßig unterbrechen und eine Fortschrittsmeldung ausgeben.

```
Function Monitor-Folder([string]$path, [System.IO.WatcherChangeTypes]$type="All",
[int]$seconds=-1) {
    $fsw = New-Object System.IO.FileSystemWatcher
    $fsw.Path = $path
    $duration = 0
    Write-Host "Überwache '$path'" -NoNewline

    while (($duration -lt $seconds) -or ($seconds -eq -1)) {
        $result = $fsw.WaitForChanged($type, 1000)
        if ($result.TimedOut -eq $FALSE) {
            Write-Host ""
            break
        } else {
            Write-Host -NoNewline "."
        }
        $duration += 1
    }
}
```

```

if ($result.TimedOut) {
    Write-Host ""
    Write-Host "Timeout erreicht." -ForegroundColor White →
    -BackgroundColor Red
}
$result
}

PS > $event = Monitor-Folder ($Home) "All" 20
PS > $event

```

Zwar unterstützt das *FileSystemWatcher*-Objekt auch asynchrone Ereignisse (Events), sodass Sie sich bei jeder überwachten Änderung informieren lassen könnten, jedoch unterstützt PowerShell in Version 1 keine Ereignisse. Ein Grund liegt an der Tatsache, dass PowerShell in einem einzelnen Thread ausgeführt wird. Trifft ein Ereignis ein, kann PowerShell es nicht bearbeiten, weil es selbst damit beschäftigt ist, den *FileSystemWatcher* auszuführen.

Sie können sich aber behelfen, indem Sie einen neuen zweiten Thread selbst erzeugen. Der einfachste Weg ist, ein Formular (Fenster) anzulegen. Wie dies geschieht, zeigt das nächste umfangreiche Beispiel. Es lädt die für Formulare und Menüs nötigen .NET-Bibliotheken und legt dann ein Formular sowie ein simples Menü an. Der entscheidende Punkt ist, dem *FileWatcherObject* das neu erzeugte Formular als *SynchronizingObject* zuzuweisen.

Wenn Sie dieses Skript ausführen, überwacht es als Vorgabe Ihren Profilordner einschließlich sämtlicher Unterordner auf beliebige Änderungen. Tritt eine Änderung ein, wird im Taskleistenbereich ein Ballonfenster geöffnet und die Änderung beschrieben. Die Änderung wird außerdem in die Konsole ausgegeben. Im Taskleistenbereich ist außerdem ein neues Symbol zu sehen. Klicken Sie mit der rechten Maustaste darauf, öffnet sich ein Menü, mit dem die Überwachung abgebrochen werden kann.

```

Function Monitor-Folder([string]$path=($Home), →
[System.IO.WatcherChangeTypes]$type="All") {
    [void][System.Reflection.Assembly]::LoadWithPartialName →
    ("System.Drawing")
    [void][System.Reflection.Assembly]::LoadWithPartialName →
    ("System.Windows.Forms")

    if ($path -eq "."){
        $path=$((Get-Item .).FullName)
    }

    $form = New-Object System.Windows.Forms.Form
    $menu = New-Object System.Windows.Forms.ContextMenu
    $menuitem = New-Object System.Windows.Forms.MenuItem
    $notifyicon = New-Object System.Windows.Forms.NotifyIcon

    $form.contextMenu = $menu
    [void]$form.contextMenu.MenuItems.Add($menuitem)
    $form.ShowInTaskbar = $FALSE
    $form.WindowState = "minimized"

```

```

$menuitem.Index = 0
$menuitem.Text = "Ü&berwachung abbrechen"
$menuitem.add_Click({
    $fsw.EnableRaisingEvents = $FALSE
    $notifyicon.Visible = $FALSE
    $form.Close()
})

# Hier den Pfad zu einem existierenden Symbol angeben:
$notifyicon.Icon = New-Object System.Drawing.Icon →
("$env:windir\system32\acwizard.ico")
$notifyicon.ContextMenu = $menu
$notifyicon.Text = "Überwache $path"

$fsw = New-Object System.IO.FileSystemWatcher $path
$fsw.IncludeSubDirectories = $TRUE
$fsw.SynchronizingObject = $form

$fsw.add_Renamed({
    $notifyicon.ShowBalloonTip(10,$_.Name,
        "$($_.OldFullPath) wurde umbenannt in $($_.FullPath)", →
        [System.Windows.Forms.ToolTipIcon]"Info")
    Write-Host "$((Get-Date).ToShortTimeString()) : →
        $($_.OldFullPath) umbenannt in $($_.FullPath)"
    })
$fsw.add_Created({
    $notifyicon.ShowBalloonTip(10,$_.Name,
        "$($_.FullPath) wurde angelegt", →
        [System.Windows.Forms.ToolTipIcon]"Info")
    Write-Host "$((Get-Date).ToShortTimeString()) : →
        $($_.FullPath) wurde angelegt"
    })
$fsw.add_Deleted({
    $notifyicon.ShowBalloonTip(10,$_.Name,
        "$($_.FullPath) wurde gelöscht", →
        [System.Windows.Forms.ToolTipIcon]"Warning")
    Write-Host "$((Get-Date).ToShortTimeString()) : →
        $($_.FullPath) wurde gelöscht"
    })
$fsw.add_Changed({
    # $notifyicon.ShowBalloonTip(10,$_.Name,
    # "Der Inhalt des Ordners $($_.FullPath) wurde verändert", →
    [System.Windows.Forms.ToolTipIcon]"Info")
    Write-Host "$((Get-Date).ToShortTimeString()) : →
        $($_.FullPath) wurde verändert"
    })

$fsw.EnableRaisingEvents = $TRUE
$notifyicon.Visible = $TRUE
[void]$form.ShowDialog()
}

```

PS > **monitor-folder**

Limitationen des *FileSystemWatchers* sind der begrenzte interne Pufferspeicher von 4 KB, der nur jeweils Raum für 50 bis 80 Änderungen bietet. Treten sehr viele Ereignisse in rascher Folge auf, können Einträge deswegen fehlen. Außerdem können Ereignisse scheinbar mehrfach auftreten, was immer dann geschieht, wenn ein Programm zum Beispiel beim Anlegen einer neuen Datei mehrfach auf die überwachten Eingabe/Ausgabe-Funktionen zugreift.

Möchten Sie die Überwachung weiter verfeinern, könnten Sie zusätzlich den *NotifyFilter* einsetzen, der dann dafür sorgt, dass nicht sämtliche Änderungen an einer Datei berücksichtigt werden, sondern nur in von Ihnen ausgewählten Bereichen:

```
PS > $fsw = New-Object System.IO.FileSystemWatcher
PS > $fsw.Path = $home
PS > $fsw.includeSubdirectories = $true
PS > $fsw.NotifyFilter = "Size,Filename"
PS > $result = $fsw.WaitForChanged("All")
PS > $result
```

ChangeType	Name	OldName	TimedOut
Renamed	img100.jpg	img25.jpg	False

In diesem Beispiel würden Dateiänderungen nur gemeldet, wenn sie die Dateigröße oder den Dateinamen betreffen. Mögliche Filtereinstellungen liefert wieder die Aufzählung der erlaubten Werte:

```
PS > [System.Enum]::GetNames([System.IO.NotifyFilters])
FileName
DirectoryName
Attributes
Size
LastWrite
LastAccess
CreationTime
Security
```

Darüber hinaus lassen sich die überwachten Dateitypen mit der *Filter*-Eigenschaft kontrollieren. Im folgenden Beispiel würden nur Textdateien in die Überwachung eingeschlossen:

```
PS > $fsw = New-Object System.IO.FileSystemWatcher
PS > $fsw.Path = $home
PS > $fsw.Filter = "*.txt"
PS > $fsw.IncludeSubdirectories = $true
PS > $result = $fsw.WaitForChanged("All", 20000)
PS > $result
```

ChangeType	Name	OldName	TimedOut
True			

ACHTUNG Wenn Sie sehr große Verzeichnisstrukturen überwachen und Filter verwenden, kann es einige Zeit dauern, bis der *FileSystemWatcher* die infrage kommenden Dateien identifiziert hat. In dieser Zeit ist es nicht möglich, PowerShell abzubrechen. Sollten Sie einen Filter verwenden, und die zu überwachende Datei ist von PowerShell bereits exklusiv geöffnet, zum Beispiel, weil Sie die Datei zuvor bereits einmal gelesen und nicht wieder ordnungsgemäß geschlossen haben, führt dies zu einem Lockdown. PowerShell reagiert dann nicht mehr und Sie müssen es neu starten.

Echtzeit-Überwachung von Änderungen an Dateien

Sie möchten einen Dateiinhalt überwachen und alarmiert werden, wenn neuer Inhalt hinzugefügt wird. Sie wollen zum Beispiel eine Logbuchdatei überwachen und jeweils nur die neuesten Einträge lesen.

Lösung

Das Cmdlet *Get-Content* verfügt über den Parameter *-wait*. Wird er angegeben, liest das Cmdlet den aktuellen Inhalt der Datei und überwacht die Datei anschließend. Sobald neue Inhalte angefügt werden, liefert *Get-Content* diese. PowerShell bricht die Ausführung erst ab, wenn Sie mit der Tastenkombination **Strg** + **C** unterbrechen. Die zu überwachende Textdatei muss bereits existieren.

```
PS > "Test" > test.txt
PS > Get-Content test.txt -wait
Test
```

Öffnen Sie die Datei *test.txt*, fügen neuen Text hinzu und speichern die Datei, liefert *Get-Content* den neu hinzugefügten Text.

Hintergrund

Der Parameter *-wait* ist leider weit weniger nützlich, als es auf den ersten Blick erscheint, denn es ist nicht möglich, nur die neu zu einer Datei hinzugefügten Daten zu lesen. Zweitens bricht die Ausführung nicht automatisch ab, zum Beispiel nach einer vorgegebenen Zeit oder nach dem Einlaufen neuer Informationen. Wird der *-wait*-Parameter also zum Beispiel innerhalb eines Skripts verwendet, blockiert das Skript an dieser Stelle und wartet ewig.

Änderungen an einer Datei können auch auf andere Weise überwacht werden. Das *FileSystemWatcher*-Objekt kann beispielsweise diese Aufgabe übernehmen. Das nächste Beispiel überwacht Änderungen an einer Datei. Sobald eine Änderung auftritt, meldet das Skript diese Änderung und setzt seine Arbeit fort:

```
$datei = "$home\test.txt"

# eine Beispieldatei anlegen:
"Test" > $datei

# Änderungen überwachen:
$fsw = New-Object System.IO.FileSystemWatcher
# Ordner überwachen, in dem sich Datei befindet:
$fsw.Path = Split-Path $datei
# Als Filter den Namen der Datei verwenden, damit nur diese
# überwacht wird:
$fsw.Filter = Split-Path $datei -leaf
$result = $fsw.WaitForChanged("All")
if ($result.TimedOut) {
    "Keine Änderungen"
} else {
    $result
}
```

Jetzt werden Sie zwar alarmiert, sobald sich die überwachte Datei ändert. Sie wissen aber nicht, in welchem Zustand die Datei vorher war. Interessieren Sie sich zum Beispiel jeweils nur für die neu hinzugefügten Informationen, müssten Sie wissen, um wie viele Bytes die Dateigröße angewachsen ist. Diese Information erhalten Sie jedoch sehr einfach, indem Sie die Dateigröße vor der Überwachung lesen und nach Eintreffen des Ereignisses mit der neuen Dateigröße vergleichen. Das nächste Beispiel demonstriert dies und verfeinert außerdem die Überwachung so, dass nur Änderungen an der Dateigröße überwacht werden:

```
Function Monitor-Filechange([string]$path) {
    if (!(Test-Path $path)) {
        Write-Error "$path existiert nicht."
        break
    } else {
        $sizeOld = (Get-ChildItem $path).length
    }
    $fsw = New-Object System.IO.FileSystemWatcher
    $fsw.Path = Split-Path $path
    $fsw.Filter = Split-Path $path -leaf
    $fsw.NotifyFilter = "Size"
    $result = $fsw.WaitForChanged("All")

    $sizeNew = (Get-ChildItem $path).length
    ($sizeNew-$sizeOld)
}

PS > Monitor-Filechange $Home\test.txt
30
```

HINWEIS

Denken Sie daran: Enthält eine Textdatei Zeichen im Unicode-Zeichensatz, werden pro Zeichen zwei Bytes verwendet, andernfalls nur eines. Fügen Sie also einer testweise überwachten Textdatei nur ein Zeichen hinzu und speichern die Änderung, meldet die Funktion möglicherweise eine Vergrößerung der Datei um

zwei Bytes, wenn es sich um Unicode-Text handelt. Wenn Sie einen Filter verwenden, stellen Sie unbedingt sicher, dass die überwachte Datei nicht bereits exklusiv von PowerShell geöffnet wurde, zum Beispiel von einem vorherigen Skript oder Test. Sobald Sie versuchen, eine bereits von Ihnen exklusiv geöffnete Datei zu überwachen, hängt PowerShell und reagiert nicht mehr.

Möchten Sie herausfinden, welche Informationen der überwachten Datei hinzugefügt wurden, benötigen Sie einen Weg, den Dateiinhalt ab einer bestimmten Position zu lesen. Nur so könnten Sie den neu hinzugefügten Inhalt lesen, ohne den vorher vorhandenen Text ebenfalls lesen zu müssen.

Leider ist *Get-Content* nicht in der Lage, Dateiinhalte ab einer bestimmten Position zu lesen, sondern liest immer von Anfang an. Greifen Sie deshalb auf das *StreamReader*-Objekt von .NET Framework zurück und verwenden Sie die *Position*-Eigenschaft des zugrunde liegenden *BaseStream*-Objekts, um die Leseposition anzupassen:

```
Function Monitor-Filechange([string]$path) {
    if (!(Test-Path $path)) {
        Write-Error "$path existiert nicht."
        break
    } else {
        $sizeOld = (Get-ChildItem $path).length
    }
    $fsw = New-Object System.IO.FileSystemWatcher
    $fsw.Path = Split-Path $path
    $fsw.Filter = Split-Path $path -leaf
    $fsw.NotifyFilter = "Size"
    $result = $fsw.WaitForChanged("All")

    $sizeNew = (Get-ChildItem $path).length
    ($sizeNew-$sizeOld)
}

Function Get-Filechange([string]$path, [int]$bytes) {
    $size = (Get-ChildItem $path).length
    $sr = New-Object System.IO.StreamReader($path)
    $sr.BaseStream.Position = ($size - $bytes)
    $sr.ReadToEnd()
    $sr.close()
}

PS > $bytes = monitor-filechange $Home\test.txt
PS > $bytes
44
PS > get-filechange $Home\test.txt $bytes
E i n e   a n g e f _ g t e   Z e i l e
```

TIPP

Auch hier kommt es wieder auf das Textformat innerhalb der Datei an. Erhalten Sie zum Beispiel Dateiänderungen als »Sperrschrift«, handelt es sich um Unicode-Text, der, wie bereits erwähnt, pro Zeichen zwei Bytes verwendet.

Ungültige Dateinamen ermitteln

Sie wollen prüfen, ob ein Dateiname ungültige Zeichen enthält.

Lösung

Rufen Sie die Liste der verbotenen Zeichen ab und vergleichen Sie sie Zeichen für Zeichen mit dem Dateinamen:

```
PS > $datei = 'log22:11-273.233.log'
PS > $verboten = @(Compare-Object ([System.IO.Path]::GetInvalidFileNameChars()) →
    ([char[]]$datei) -IncludeEqual -ExcludeDifferent -PassThru)
PS > If ($verboten.Count) {
>> "Sie haben die verbotenen Zeichen '$verboten' in '$datei' verwendet"
>> }
>>
```

Hintergrund

Die Systemfunktion `[System.IO.Path]::GetInvalidFileNameChars()` liefert automatisch eine Liste von Zeichen, die in Dateinamen nicht erlaubt sind. Um zu überprüfen, ob diese Zeichen in einem Dateinamen vorkommen, verwandelt man auch den Dateinamen in eine solche Liste, indem der Text in ein Array von einzelnen Zeichen umgewandelt wird (`[Char[]]`).

Man hat also nun zwei Listen. Die erste enthält verbotene Zeichen, die zweite enthält die Zeichen des zu überprüfenden Dateinamens. Mit *Compare-Object* können beide Listen nun miteinander verglichen werden. Da die Überprüfung nur Zeichen finden soll, die in beiden Listen vorkommen, werden dabei die Parameter `-ExcludeDifferent` und `-IncludeEqual` eingesetzt. Mit dem Parameter `-passThru` sorgt man dafür, dass *Compare-Object* die in beiden Listen vorhandenen Zeichen wieder zurückgibt.

Das Ergebnis ist in `$verboten` eine Liste von verbotenen Zeichen, die im Dateinamen aber vorkommen. Enthält die Liste verbotene Zeichen, gibt der Code eine Warnmeldung aus und listet die verbotenen Zeichen auf.

Zusammenfassung

PowerShell bezeichnet den Inhalt von Datenträgern als *Item*. Eine ganze Armada von Cmdlets steht bereit, um mit diesen Items umzugehen. Sie finden die Namen dieser Cmdlets mit folgendem Aufruf:

```
PS > Get-Command -Noun Item*
```

CommandType	Name	Definition
Cmdlet	Clear-Item	Clear-Item [-Path] <String[]> [-For...
Cmdlet	Clear-ItemProperty	Clear-ItemProperty [-Path] <String[...
Cmdlet	Copy-Item	Copy-Item [-Path] <String[]> [[-Des...
Cmdlet	Copy-ItemProperty	Copy-ItemProperty [-Path] <String[]...
(...)		

Mit diesen Cmdlets können Sie nun Dateien und Ordner verwalten. *Get-ChildItem* listet beispielsweise die »Kind«-Items eines anderen Items auf, also einen Ordnerinhalt. *Rename-Item* benennt ein Item um, also eine Datei oder einen Ordner. Und *Copy-Item* kopiert ein Item, sodass Sie Dateien oder ganze Ordner kopieren können.

Die Namen dieser Cmdlets wirken anfangs fremd, was an den Begrifflichkeiten liegt. Weil PowerShell sein Laufwerkskonzept nicht auf das klassische Dateisystem beschränkt, können die Cmdlet-Namen nicht eingängigere Begriffe wie »File« oder »Directory« verwenden. Deshalb gibt es historische Aliasnamen wie *dir*, *cd* oder *del*, über die Sie die zuständigen Cmdlets ebenfalls erreichen:

```
PS > Get-Alias -definition *-Item*
```

CommandType	Name	Definition
Alias	cli	Clear-Item
Alias	clp	Clear-ItemProperty
Alias	copy	Copy-Item
Alias	cp	Copy-Item
Alias	cpi	Copy-Item
Alias	cpp	Copy-ItemProperty
Alias	del	Remove-Item
Alias	erase	Remove-Item
Alias	gi	Get-Item
Alias	gp	Get-ItemProperty
Alias	ii	Invoke-Item
Alias	mi	Move-Item
Alias	move	Move-Item
Alias	mp	Move-ItemProperty
Alias	mv	Move-Item
Alias	ni	New-Item
Alias	rd	Remove-Item
Alias	ren	Rename-Item
Alias	ri	Remove-Item
Alias	rm	Remove-Item
Alias	rmdir	Remove-Item
Alias	rni	Rename-Item
Alias	rnp	Rename-ItemProperty
Alias	rp	Remove-ItemProperty
Alias	si	Set-Item
Alias	sp	Set-ItemProperty

Der eigentliche Inhalt einer Datei wird von Cmdlets aus der Familie »Content« verwaltet, mit deren Hilfe Sie den Inhalt von Dateien lesen, ändern oder erweitern:

PS > **Get-Command -noun Content**

CommandType	Name	Definition
-----	----	-----
Cmdlet	Add-Content	Add-Content [-Path] <String[]> [-Value] <0...
Cmdlet	Clear-Content	Clear-Content [-Path] <String[]> [-Filter ...
Cmdlet	Get-Content	Get-Content [-Path] <String[]> [-ReadCount...
Cmdlet	Set-Content	Set-Content [-Path] <String[]> [-Value] <0...

Zusätzlich steht das Cmdlet *Out-File* zur Verfügung, das die meisten Aufgaben rund um das Anlegen von textbasierten Dateien am einfachsten abwickelt.

Um auf Dateien und Ordner zuzugreifen, gibt man ihren Pfadnamen an. Dies kann absolut oder relativ geschehen und die Platzhalterzeichen für relative Pfadnamen listet Tabelle 7.1 (siehe Seite 214) auf. Wollen Sie selbst Pfade generieren oder überprüfen, greifen Sie zur Familie der *Path*-Cmdlets:

PS > **Get-Command -noun Path**

CommandType	Name	Definition
-----	----	-----
Cmdlet	Convert-Path	Convert-Path [-Path] <String[]> [-Verbose]...
Cmdlet	Join-Path	Join-Path [-Path] <String[]> [-ChildPath] ...
Cmdlet	Resolve-Path	Resolve-Path [-Path] <String[]> [-Relative...
Cmdlet	Split-Path	Split-Path [-Path] <String[]> [-LiteralPat...
Cmdlet	Test-Path	Test-Path [-Path] <String[]> [-Filter <Str...

Zusätzlich stehen für diese Aufgaben Low-Level-Funktionen von .NET Framework zur Verfügung, die Tabelle 7.2 auf Seite 220 aufführt.

Der aktuelle Pfad, der die Grundlage relativer Pfadnamen bildet, wird von Cmdlets aus der Familie *Location* verwaltet. Mit diesen Cmdlets können Sie also den aktuellen Ort ermitteln, ändern oder vorübergehend zwischenspeichern (*Push-Location*) und später wiederherstellen (*Pop-Location*):

PS > **Get-Command -noun Location**

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Location	Get-Location [-PSProvider <String[]>] [-PS...
Cmdlet	Pop-Location	Pop-Location [-PassThru] [-StackName <Stri...
Cmdlet	Push-Location	Push-Location [[-Path] <String>] [-PassThr...
Cmdlet	Set-Location	Set-Location [[-Path] <String>] [-PassThru...

Nicht immer sind die PowerShell-Cmdlets der beste Weg, um eine Aufgabe zu lösen. Insbesondere umfangreiche Dateisystem-Kopieraufgaben löst man besser, schneller und zuverlässiger mit den bewährten Konsolenbefehlen wie *xcopy* oder *robocopy*. Sie lassen sich nahtlos in PowerShell-Code einbinden.

Kapitel 8

Registrierungsdatenbank

In diesem Kapitel:

Alle Unterschlüssel eines Schlüssels lesen	265
Neuen Registrierungsschlüssel anlegen	268
Prüfen, ob ein Schlüssel existiert	271
Bestimmten Schlüssel suchen	272
Schlüssel löschen	273
Wert eines Schlüssels lesen	274
Standardwert eines Schlüssels lesen	278
Werte vieler Schlüssel auslesen	278
Prüfen, ob ein bestimmter Wert existiert	279
Wert eines Schlüssels ändern	280
Wert eines Schlüssels löschen	282
Standardwert eines Schlüssels löschen	284
Windows-Registrierungs-Editor öffnen	285
In der Registrierungsdatenbank navigieren	286
Andere Orte der Registrierungsdatenbank ansprechen	288

Remote auf Registrierungsdatenbank zugreifen	289
Zusammenfassung	291

Die Windows-Registrierungsdatenbank ist der zentrale Speicher für alle Windows-Einstellungen. Anwendungen speichern darin ihre Konfiguration, Windows hinterlegt hier Hard- und Softwareinformationen und Administratoren greifen häufig auf die Registrierungsdatenbank zu, um Fragestellungen zur Systemkonfiguration zu klären oder selbst einzugreifen und Änderungen vorzunehmen.

Weil PowerShell den Inhalt der Registrierungsdatenbank als Laufwerk zur Verfügung stellt, navigieren Sie in der Registrierungsdatenbank mit genau denselben Cmdlets wie im Dateisystem und können ähnlich einfach Registrierungsschlüssel lesen, ändern, anlegen oder löschen. Dabei entsprechen die Registrierungsschlüssel den Items (siehe auch vergangenes Kapitel). Die Werte der Registrierungsschlüssel dagegen werden als *ItemProperty* bezeichnet und von Cmdlets der Familie *ItemProperty* verwaltet.

Der Grundaufbau der Registrierungsdatenbank wird aus Abbildung 8.1 deutlich:

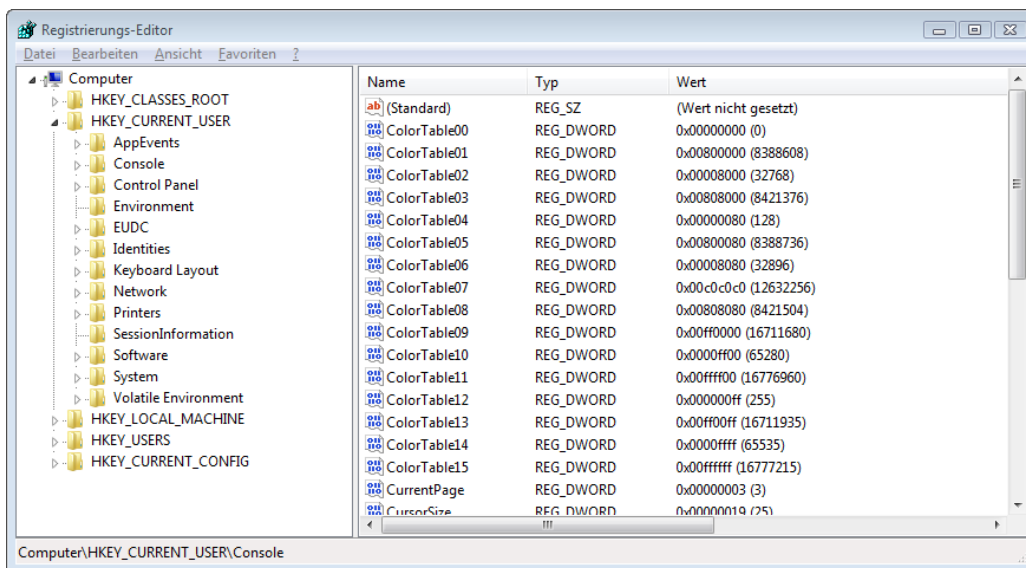


Abbildung 8.1 Die Registrierungsdatenbank aus Sicht des Registrierungs-Editors

- **Wurzelzweige** In der linken Spalte sind in Großbuchstaben die Wurzelzweige der Registrierungsdatenbank zu sehen. *HKEY_CURRENT_USER* speichert die Benutzerdaten des aktuell angemeldeten Benutzers, *HKEY_LOCAL_MACHINE* speichert alle allgemeinen Konfigurationsdaten des Computers und darf deshalb in aller Regel nur von Administratoren verändert werden.

- **Schlüssel** In der linken Spalte sieht man in Form gelber Ordner die Registrierungsschlüssel (kurz Schlüssel). Sie funktionieren tatsächlich so ähnlich wie Dateiordner und können Dateien (hier Werte genannt) und Unterordner (weitere Schlüssel) enthalten. PowerShell bezeichnet sie als Item.
- **Werte und Datentypen** Der Inhalt eines Schlüssels wird in der rechten Spalte gezeigt: die Werte. Jeder Wert trägt einen Namen und speichert Informationen, die in der Spalte *Wert* zu sehen sind. Die Informationen können in unterschiedlichen Datentypen gespeichert werden. Jedem Wert ist ein ganz bestimmter Datentyp zugewiesen, der in der Spalte *Typ* genannt wird. PowerShell bezeichnet sie als ItemProperty.
- **Standardwert** Jeder Schlüssel kann einen sogenannten Standardwert besitzen. Der Standardwert wird in der rechten Spalte an oberster Stelle unter dem Namen (*Standard*) geführt. In Wirklichkeit hat dieser Wert überhaupt keinen Namen. Er ist auch für die meisten Schlüssel nicht gesetzt. Der Registrierungs-Editor meldet dann in der Spalte *Wert* den Hinweis (*Wert nicht gesetzt*).

ACHTUNG Die Registrierungsdatenbank speichert wesentliche Konfigurationsdaten Ihres Computers. Änderungen werden sofort wirksam. Fehlerhafte Einträge und unbedachtes Löschen von Einträgen können Ihren Computer beschädigen und unbrauchbar machen. Schlimmstenfalls muss Windows dann neu installiert werden. Weil Automationswerkzeuge wie PowerShell nicht nur die erwünschten Aktionen beschleunigen, sondern auch eventuell fehlerhafte, beachten Sie bitte:

Sobald Sie Cmdlets einsetzen, die nicht das Verb »Get« verwenden (und also lesenderweise stets harmlos sind),

- führen Sie keine Änderungen unbedacht durch, nur um zu sehen, was als Nächstes geschieht
- arbeiten Sie möglichst immer ohne Administratorrechte
- trainieren Sie niemals an Produktivsystemen
- legen Sie ein Backup an, bevor Sie umfangreichere Änderungen vornehmen

Alle Unterschlüssel eines Schlüssels lesen

Sie möchten eine Liste sämtlicher Unterschlüssel erstellen, die in einem Schlüssel enthalten sind.

Lösung

Weil PowerShell den Inhalt der Registrierungsdatenbank als Laufwerk interpretiert, werden Registrierungsschlüssel wie Ordner interpretiert. Mit *Get-ChildItem* lesen Sie den Inhalt eines Registrierungsschlüssels und erhalten eine Liste seiner Unterschlüssel:

```
PS > Get-ChildItem HKLM:\Software
```

```
Hive: Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software
```

SKC	VC	Name	Property
---	---	---	-----
1	0	Acrobat Reader	{}
1	0	Actipro Software	{}
2	0	Adobe	{}
(...)			

Die Spalte *SKC* (Subkey Count) meldet, wie viele Unterschlüssel sich in einem Schlüssel befinden. Die Spalte *VC* (Value Count) gibt die Anzahl der Werte eines Schlüssels an. Die Namen der Werte werden in der Spalte *Property* genannt.

Möchten Sie auch die Unterschlüssel der Unterschlüssel erfassen, also rekursiv suchen, geben Sie zusätzlich den Parameter *-recurse* an:

```
PS > Get-ChildItem HKLM:\Software -recurse -ErrorAction SilentlyContinue
```

Anstelle des virtuellen Laufwerks darf auch der tatsächliche Registrierungsschlüssel angegeben werden, wenn Sie davor den Namen des für die Registrierungsdatenbank zuständigen Providers anfügen:

```
PS > Get-ChildItem Registry::HKEY_LOCAL_MACHINE\Software -recurse →  
-ErrorAction SilentlyContinue
```

Hintergrund

Registrierungsschlüssel lesen Sie entweder über eines der virtuellen Laufwerke (*HKLM:* oder *HKCU:*). Oder Sie geben den tatsächlichen Pfadnamen eines Registrierungsschlüssels an und fügen davor »Registry::« ein. Auf diese Weise haben Sie Zugriff auf alle Zweige der Registrierungsdatenbank und nicht nur auf die beiden Zweige, die von den virtuellen Laufwerken zugänglich gemacht werden.

Die Schlüssel in einem bestimmten Zweig der Registrierungsdatenbank lesen Sie mit *dir* (*Get-ChildItem*), ähnlich dem Inhalt eines Ordners im Dateisystem. *Get-ChildItem* unterstützt Platzhalterzeichen, sodass Sie die Liste der Unterschlüssel filtern könnten. Die folgenden Zeilen liefern alle registrierten Dateierweiterungen, die sich im Schlüssel *HKEY_LOCAL_MACHINE\Software\Classes* befinden und deren Schlüsselnamen stets mit einem Punkt beginnen:

```
PS > $key = 'HKLM:\Software\Classes\.*'
PS > dir $key -name

.aca
.acdda
.acddb
.accdc
(...)
```

Die Werte eines Schlüssels (im Registrierungs-Editor werden diese in der rechten Spalte angezeigt) werden als sogenannte *ItemProperties* bezeichnet. Wie Sie die Werte eines Schlüssels lesen, wird in separaten Rezepten in diesem Kapitel gezeigt. Kombinieren Sie beide Möglichkeiten, lassen sich Funktionen wie die folgende herstellen, die nach Eingabe einer Dateierweiterung das hierfür zuständige Programm ermittelt:

```
Function Get-FileEXE([string]$erweiterung = →
$(Throw 'Dateierweiterung angeben!')) {
# Registrierungsschlüssel für Dateierweiterung finden
$key = "HKLM:\SOFTWARE\Classes\$erweiterung"
if (!(Test-Path $key)) {
    Throw "Dateierweiterung $erweiterung ist unbekannt."
}

# Registrierungsschlüssel öffnen
$regkey = Get-Item $key

# zuständiges Programm wird im Standardeintrag genannt
$wert = $regkey.GetValue('')
$prgkey = "HKLM:\SOFTWARE\Classes\$wert\shell\open\command"

if (!(Test-Path $prgkey)) {
    Throw "Dateierweiterung $erweiterung ist keinem Programm zugeordnet."
} else {
    return (Get-ItemProperty $prgkey).'(default)'
}
}
```

Sobald Sie das Skript erstellt und ausgeführt haben, steht Ihnen ein neuer Befehl namens *Get-FileEXE* zur Verfügung, der aus der Registrierungsdatenbank ausliest, welches Programm für eine bestimmte Dateierweiterung zuständig ist:

```
PS > Get-FileEXE .pdf
"C:\Program Files (x86)\Adobe\Reader 9.0\Reader\AcroRd32.exe" "%1"
PS > Get-FileEXE .vbs
"C:\Windows\System32\WScript.exe" "%1" %*
PS > Get-FileEXE .html
"C:\Program Files (x86)\Internet Explorer\iexplore.exe" -nohome
PS > Get-FileEXE .bmp
C:\Windows\System32\rundll32.exe "C:\Program Files\Windows Photo Viewer\PhotoViewer.dll",
ImageView_Fullscreen %1
```

Neuen Registrierungsschlüssel anlegen

Sie möchten einen neuen Registrierungsschlüssel in die Registrierungsdatenbank eintragen.

Lösung

Legen Sie neue Schlüssel genauso an wie neue Ordner im Dateisystem. Greifen Sie zum Beispiel zur Funktion *md*. Oder verwenden Sie das zugrunde liegende Cmdlet *New-Item* direkt:

```
PS > md HKCU:\Software\Testschlüsse1

Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
```

SKC	VC Name	Property
---	----	-----
0	0 Testschlüsse1	{}

```
PS > New-Item HKCU:\Software\Testschlüsse2

Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
```

SKC	VC Name	Property
---	----	-----
0	0 Testschlüsse2	{}

Auf diese Weise wird ein neuer, aber vollkommen leerer Schlüssel hinzugefügt. Der Schlüssel enthält keine Werte und auch sein Standardwert ist nicht definiert. Möchten Sie einen neuen Schlüssel mit einem festgelegten Standardwert anlegen, gehen Sie so vor:

```
PS > New-Item -type String HKCU:\Software\Testschlüsse13 →
-value "Standardwert ist festgelegt"

Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
```

SKC	VC Name	Property
---	----	-----
0	1 Testschlüsse13	{(default)}"

Hintergrund

Der Befehl *md* ist in Wirklichkeit eine vordefinierte Funktion, die intern das Cmdlet *New-Item* mit dem Parameter *-type directory* aufruft. Sie können neue Registrierungsschlüssel also wahlweise mit *md* oder mit *New-Item -type directory* anlegen, sofern Sie einen gültigen Registrierungsschlüssel angeben und der angegebene Schlüssel noch nicht existiert. Der Parameter *-type* muss nicht unbedingt angegeben werden, wenn Sie lediglich einen leeren neuen Schlüssel erstellen wollen, weil es in der Registrierungsdatenbank anders als im Dateisystem nur eine Sorte von Items gibt.

Als Ergebnis wird der neu angelegte Schlüssel an Sie zurückgeliefert. Sofern Sie den zurückgelieferten Schlüssel nicht speichern, wird er in die Konsole ausgegeben. Um die Ausgabe in die Konsole zu verhindern, leiten Sie das Ergebnis weiter an *Out-Null*:

```
PS > md HKCU:\Software\Testschlüsse14 | Out-Null
```

Sie können den Schlüssel aber auch in einer Variablen speichern und dann später auf die Eigenschaften und Methoden des Schlüssels zugreifen, zum Beispiel, um dem Schlüssel sofort weitere Werte zuzuweisen:

```
PS > $key = md HKCU:\Software\Testschlüsse15
PS > $key.SetValue("", "Der Standardwert")
PS > $key.SetValue("Wert1", "Ein neuer Wert")
PS > $key.SetValue("Wert2", [byte[]](1,2,3,4,5) )
PS > $key.SetValue("Wert3", [string[]]('Erste Zeile', 'Zweite Zeile'))
PS > $key.SetValue("Wert4", "Windows-Ordner: %WINDIR%", "ExpandString")
PS > $key.Close()
PS > Get-ItemProperty HKCU:\Software\Testschlüsse15

PSPath      :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Testschlüsse15
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
PSChildName  : Testschlüsse15
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
(default)    : Der Standardwert
Wert1        : Ein neuer Wert
Wert2        : {1, 2, 3, 4...}
Wert3        : {Erste Zeile, Zweite Zeile}
Wert4        : Windows-Ordner: C:\Windows
```

Den Standardwert des neuen Schlüssels können Sie auch festlegen, indem Sie *New-Item* mit dem Parameter *-type [Datentyp]* verwenden. Hier bezieht sich der Typ also nicht auf den Schlüssel, den Sie anlegen, sondern auf den Datentyp seines Standardwerts.

Mit dem Parameter *-value* legen Sie dann den Inhalt des Standardwerts fest. Im Windows-Registrierungs-Editor wird der Standardwert eines Schlüssels stets in der rechten Spalte mit dem Namen (*Standard*) genannt. Ist er nicht festgelegt, wird als Wert (*Wert nicht gesetzt*) angezeigt, andernfalls der hinterlegte Wert.

Der Parameter *-type* unterstützt in der Registrierungsdatenbank die Optionen aus Tabelle 8.1:

Typ	Beschreibung
<i>String</i>	Ein einzeliger Text
<i>ExpandString</i>	Ein einzeliger Text, bei dem Umgebungsvariablen automatisch aufgelöst werden
<i>Binary</i>	Ein Binärwert

Tabelle 8.1 Erlaubte Datentypen in der Windows-Registrierungsdatenbank

Typ	Beschreibung
<i>DWord</i>	Eine Ganzzahl (32 Bit)
<i>MultiString</i>	Ein mehrzeiliger Text
<i>QWord</i>	Eine Ganzzahl (64 Bit)

Tabelle 8.1 Erlaubte Datentypen in der Windows-Registrierungsdatenbank (*Fortsetzung*)

```

PS > # ExpandString:
PS > New-Item -type ExpandString HKCU:\Software\Testschlüsse16 →
    -value "Windows-Ordner: %windir%" > $null
PS > (Get-ItemProperty HKCU:\Software\Testschlüsse16).'(default)'
Windows-Ordner: C:\Windows
PS > # Binary
PS > New-Item -type Binary HKCU:\Software\Testschlüsse17 -value 1,2,3,4,5,6 →
    > $null
PS > (Get-ItemProperty HKCU:\Software\Testschlüsse17).'(default)'
1
2
3
4
5
6
PS > # Dword
PS > New-Item -type DWord HKCU:\Software\Testschlüsse18 -value 1024 > $null
PS > (Get-ItemProperty HKCU:\Software\Testschlüsse18).'(default)'
1024
PS > # MultiString
PS > New-Item -type MultiString HKCU:\Software\Testschlüsse19 →
    -value 'Ein mehrzeiliger Text', 'zweite Zeile' > $null
PS > (Get-ItemProperty HKCU:\Software\Testschlüsse19).'(default)'
Ein mehrzeiliger Text
zweite Zeile
PS > # Qword
PS > New-Item -type QWord HKCU:\Software\Testschlüsse10 -value 765754543256 →
    > $null
PS > (Get-ItemProperty HKCU:\Software\Testschlüsse10).'(default)'
765754543256

```

Anders als im Dateisystem ist es in der Registrierungsdatenbank nicht möglich, mit einer Anweisung gleich mehrere Schlüssel anzulegen. Der jeweils übergeordnete Registrierungsschlüssel muss bereits vorhanden sein.

TIPP

Wenn Sie den Beispielen gefolgt sind, haben Sie eine Reihe von Testschlüsseln angelegt. Um diese Testschlüssel wieder zu entfernen, genügt die Eingabe des folgenden Befehls:

```
PS > del HKCU:\Software\Testschlüss* -confirm
```

Prüfen, ob ein Schlüssel existiert

Sie wollen herausfinden, ob ein bestimmter Schlüssel bereits vorhanden ist.

Lösung

Prüfen Sie mit *Test-Path*, ob der Schlüssel existiert. Falls ja, wird *\$true* zurückgegeben, sonst *\$false*:

```
PS > Test-Path HKCU:\Software\Testschlüssel
False
PS > md HKCU:\Software\Testschlüssel > $null
PS > Test-Path HKCU:\Software\Testschlüssel
True
PS > del HKCU:\Software\Testschlüssel
Abhängig vom Ergebnis könnten Sie beispielsweise einen Schlüssel anlegen, falls er noch nicht existiert:
if (!(Test-Path HKCU:\Software\Testschlüssel)) { md HKCU:\Software\Testschlüssel | Out-Null }
```

Hintergrund

Test-Path prüft, ob der angegebene Pfad vorhanden ist, und funktioniert in der Registrierungsdatenbank genauso wie im Dateisystem. So könnten Sie zum Beispiel eine Funktion realisieren, die beliebig verschachtelte Registrierungsschlüssel in einem Schritt anlegt.

Stellen Sie *\$DebugPreference* vorübergehend auf *'Continue'* ein, um mithilfe der Debugmeldungen besser verstehen zu können, wie die folgende Funktion arbeitet. Mit *\$DebugPreference = 'SilentlyContinue'* schalten Sie den Debugmodus wieder aus.

```
Function New-RegistryKey([string]$key) {
    # Schlüssel zerlegen
    $subkeys = $key.Split("\")
    foreach ($subkey in $subkeys) {
        $currentkey += ($subkey + '\')
        if (!(Test-Path $currentkey)) {
            md $currentkey | Out-Null
            Write-Debug "Schlüssel angelegt: $currentkey"
        }
    }
}
```

Diese Funktion legt den angegebenen Schlüssel sowie außerdem alle eventuell noch fehlenden zusätzlichen Schlüssel an:

```

PS > $DebugPreference = 'Continue'
PS > New-RegistryKey ->
    'HKCU:\Software\Testschlüssel\Bereich1\Unterbereich\Noch ein Bereich'
DEBUG: Schlüssel angelegt: HKCU:\Software\Testschlüssel\Bereich1\
DEBUG: Schlüssel angelegt: HKCU:\Software\Testschlüssel\Bereich1\Unterbereich\
DEBUG: Schlüssel angelegt: HKCU:\Software\Testschlüssel\Bereich1\Unterbereich\Noch ein
Bereich\

```

Bestimmten Schlüssel suchen

Sie möchten alle Schlüssel finden, die einen bestimmten Namen tragen.

Lösung

Durchsuchen Sie die Registrierungsdatenbank rekursiv mit *Get-ChildItem* und dem Parameter *-recurse* und geben Sie dabei Ein- und Ausschlusskriterien an:

```

PS > dir HKCU:\, HKLM:\ -recurse -include *PowerShell* -ea SilentlyContinue

Hive:
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Classes\VirtualStore
\MACHINE\SOFTWARE\Microsoft

SKC  VC Name                Property
---  ---
1    0 PowerShell            {}

Hive:
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Classes\VirtualStore
\MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds

SKC  VC Name                Property
---  ---
0    1 Microsoft.PowerShell  {ExecutionPolicy}

(...)

```

Hintergrund

Get-ChildItem unterstützt die rekursive Suche und kann dabei ein oder mehrere Laufwerke durchsuchen. Im Beispiel durchsucht die Anweisung sowohl das Laufwerk *HKCU:* als auch das Laufwerk *HKLM:*, also die Registrierungszweige *HKEY_CURRENT_USER* und *HKEY_LOCAL_MACHINE*. Wenn Sie sicherstellen wollen, dass die angegebenen Laufwerke komplett durchsucht werden, geben Sie dahinter einen umgekehrten Schrägstrich an.

Ohne diesen Schrägstrich beginnt die Suche im angegebenen Laufwerk an der aktuellen Position. Normalerweise ist die aktuelle Position die Wurzel des Laufwerks, aber wenn Sie die aktu-

elle Position des Laufwerks mit *cd* (*Set-Location*) geändert haben, sind die Suchergebnisse ohne Angabe des umgekehrten Schrägstrichs unvollständig:

```
PS > cd HKLM:\System\Setup
PS > dir HKLM: -recurse -include *PowerShell*
PS > dir HKLM:\ -recurse -include *PowerShell*
```

Der Registry-Provider unterstützt im Gegensatz zum FileSystem-Provider keine Filter. Sie müssen deshalb für die Suche immer die Option *-include* angeben. Ohne diese Option wird Ihr Suchwort als Filter verstanden. Sie erhalten dann einen Fehler:

```
PS > dir HKLM:\ -recurse *PowerShell*
Get-ChildItem : Die Methode kann nicht aufgerufen werden. Der Anbieter unterstützt keine
Verwendung von Filtern.
Bei Zeile:1 Zeichen:4
+ dir <<<< HKLM:\ -recurse *PowerShell*
```

Fehler erhalten Sie auch, wenn die Suche Registrierungsschlüssel ergab, auf die Sie keine Zugriffsberechtigungen haben. Möchten Sie solche Fehlermeldungen unterdrücken, stellen Sie die *ErrorAction* auf *SilentlyContinue* ein:

```
PS > dir HKLM:\ -recurse -include *PowerShell* -ea SilentlyContinue
```

Schlüssel löschen

Sie möchten einen vorhandenen Registrierungsschlüssel mit allen seinen Unterschlüsseln und Werten entfernen.

Lösung

Vorhandene Schlüssel werden mit *del* (*Remove-Item*) entfernt:

```
PS > del HKCU:\Software\Testschlüssel1
```

Sie können auf diese Weise auch mehrere Schlüssel löschen, indem Sie Platzhalterzeichen verwenden. Greifen Sie am besten direkt zu *Remove-Item*, auf das der Alias *del* verweist:

```
PS > Remove-Item HKCU:\Software\Testschlüssel* -confirm
```

Enthält der Schlüssel Unterschlüssel, erscheint eine Sicherheitsabfrage, bevor der Schlüssel mit- samt seiner Unterschlüssel entfernt wird. Möchten Sie die Sicherheitsabfrage umgehen, verwenden Sie den Parameter *-recurse*:

```
PS > Remove-Item HKCU:\Software\Testschlüssel -recurse
```

Hintergrund

del (Kurzform für *Remove-Item*) entfernt Schlüssel nur dann sofort, wenn die Schlüssel keine weiteren Unterschlüssel enthalten. Damit Sie nicht unbeabsichtigt Schlüssel mitsamt Unterschlüssel entfernen, erscheint beim Löschen von Schlüsseln, die Unterschlüssel enthalten, eine Sicherheitsabfrage, die zuerst mit »J« für »Ja« beantwortet werden muss.

```
PS > md HKCU:\Software\Testschlüssel20 > $null
PS > md HKCU:\Software\Testschlüssel20\Unterschlüssel > $null
PS > del HKCU:\Software\Testschlüssel20
```

Bestätigung

Das Element unter "HKCU:\Software\Testschlüssel20" verfügt über untergeordnete Elemente, und der Recurse-Parameter wurde nicht angegeben. Wenn Sie fortfahren, werden mit dem Element auch alle

untergeordneten Elemente entfernt. Möchten Sie den Vorgang wirklich fortsetzen?

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe (Der Standardwert ist "J"): j

Möchten Sie die Sicherheitsabfrage umgehen, was zum Beispiel für die unbeaufsichtigte Ausführung von Skripts nötig ist, verwenden Sie den Parameter *-recurse*:

```
PS > del HKCU:\Software\Testschlüssel20 -recurse
```

Remove-Item unterstützt Platzhalterzeichen, sodass Sie mit einer Anweisung mehrere Schlüssel entfernen können. Möglicherweise gelten Ihre Platzhalterzeichen allerdings für weit mehr Schlüssel, als Sie gedacht haben. Verwenden Sie Platzhalterzeichen deshalb nur zusammen mit den Optionen *-whatIf* oder *-confirm*, damit Sie die Operation zuerst überprüfen und gegebenenfalls einzeln bestätigen können.

Wert eines Schlüssels lesen

Sie möchten den Wert eines einzelnen Registrierungsdatenbank-Schlüssels auslesen.

Lösung

Verwenden Sie das Cmdlet *Get-ItemProperty* und fragen Sie den gewünschten Wert daraus ab. Die folgende Zeile liest den Wert *ProxyEnable* aus dem Schlüssel *HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings*:

```
PS > $schluesel = →
    'HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings'
PS > $wert = 'ProxyEnable'
PS > (Get-ItemProperty $schluesel).$wert
1
```

Sie könnten dies auch in einer einzelnen Zeile tun:

```
PS > (Get-ItemProperty -Path
      'HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings') -Name
      .ProxyEnable
1
```

Alternativ kann man anstelle von *Get-ItemProperty* auch *Get-Item* verwenden, um zuerst auf den Registrierungsschlüssel zuzugreifen, und dann mit der Methode *GetValue()* einen bestimmten Wert daraus lesen:

```
PS > $schluessel = Get-Item -Path
      'HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings'
PS > $wert = $schluessel.GetValue('ProxyEnable')
PS > $wert
1
```

In einer Zeile sieht dieser Ansatz folgendermaßen aus:

```
PS > (Get-Item -Path
      'HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings')
      .GetValue('ProxyEnable')
1
```

Hintergrund

Get-ItemProperty liefert sämtliche Werte des angegebenen Registrierungsschlüssels. Jeder einzelne Wert kann dabei wie eine Eigenschaft abgerufen werden:

```
PS > $pfad = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings"
PS > $alles = Get-ItemProperty $pfad
PS > $alles

PSPath                :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\...
PSParentPath          :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\...
PSChildName           : Internet Settings
PSDrive               : HKCU
PSProvider             : Microsoft.PowerShell.Core\Registry
IE5_UA_Backup_Flag    : 5.0
User Agent            : Mozilla/4.0 (compatible; MSIE 7.0; Win32)
EmailName             : IEUser@
AutoConfigProxy       : wininet.dll
MimeExclusionListForCache : multipart/mixed multipart/x-mixed-replace multipart/x-
byteranges
(...)

PS > $alles.ProxyEnable
1
```

PowerShell fügt zu den Registrierungswerten fünf eigene Eigenschaften hinzu, deren Name jeweils mit »PS« beginnt. Wollen Sie diese Informationen ausblenden, verwenden Sie *Select-Object* mit dem Parameter *-exclude*. Jetzt allerdings würden alle Werte ausgeblendet, die mit »PS« beginnen:

```
PS > Get-ItemProperty $pfad | Select-Object * -exclude 'PS*'
```

HINWEIS

Zwar verfügt auch *Get-ItemProperty* über einen Parameter namens *-exclude*, doch gilt dieser nur für tatsächlich vorhandene Werte. Mit ihm kann man also die nachträglich von PowerShell hinzugefügten Werte nicht entfernen.

Der Standardwert des Registrierungsschlüssels wird unter dem besonderen Namen »(default)« aufgeführt. Möchten Sie auf Werte zugreifen, die Leerzeichen oder andere Sonderzeichen im Namen tragen (beispielsweise auf den Wert »(default)«), setzen Sie den Namen in Anführungszeichen:

```
PS > $alles.'User Agent'
Mozilla/4.0 (compatible; MSIE 7.0; Win32)
PS > $alles.'(default)'
PS >
```

Wenn Sie nur den Inhalt eines einzelnen Werts lesen wollen, sparen Sie Speicherplatz, indem Sie *Get-ItemProperty* von vornherein auf diesen einzelnen Wert beschränken, sodass nicht sämtliche Werte des Schlüssels abgerufen werden müssen:

```
PS > $pfad = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings"
PS > $alles = Get-ItemProperty $pfad 'User Agent'
PS > $alles

PSPath      : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\...
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\...
PSChildName  : Internet Settings
PSDrive      : HKCU
PSProvider   : Microsoft.PowerShell.Core\Registry
User Agent   : Mozilla/4.0 (compatible; MSIE 7.0; Win32)

PS > $alles.'User Agent'
Mozilla/4.0 (compatible; MSIE 7.0; Win32)
```

Möchten Sie mehrere Werte erfragen, ist es effizienter, den Schlüssel in einer Variablen zwischenspeichern, als ihn mehrmals hintereinander zu öffnen.

Um einen bestimmten Wert eines Schlüssels abzurufen, können Sie das Ergebnis von *Get-ItemProperty* in einer Variablen speichern und dann die gewünschte Eigenschaft über die Punkt-Schreibweise daraus abrufen. Das kann auch in einer einzelnen Zeile geschehen:

```
PS > (Get-ItemProperty $pfad 'User Agent').'User Agent'
Mozilla/4.0 (compatible; MSIE 7.0; Win32)
```

Grundsätzlich dürfen Sie auch auf Registrierungsschlüssel direkt und unter Umgehung eines virtuellen Laufwerks zugreifen, wenn Sie vor den Registrierungspfad den Namen des zuständigen Providers schreiben. Die folgenden beiden Zeilen liefern dasselbe Resultat:

```
PS > Get-ItemProperty →
    'HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings'
PS > Get-ItemProperty 'Registry →
    ::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings'
```

Sie erreichen die Werte eines Registrierungsschlüssels auch über den Registrierungsschlüssel selbst. Dazu sprechen Sie den Registrierungsschlüssel mit *Get-Item* an. Jeder Registrierungsschlüssel besitzt eine *GetValue()*-Methode, über die man gezielt einen seiner Werte erfragen kann.

```
PS > (Get-Item $pfad).GetValue('User Agent')
Mozilla/4.0 (compatible; MSIE 7.0; Win32)
```

Dieser Ansatz liefert auf Wunsch weitere Informationen über den Wert, beispielsweise seinen Datentyp:

```
PS > (Get-Item $pfad).GetValueKind('User Agent')
String
```

Auch erhalten Sie so auf Wunsch eine Liste der Namen sämtlicher Unterschlüssel:

```
PS > (Get-Item $pfad).GetValueNames()
IE5_UA_Backup_Flag
User Agent
EmailName
(...)
```

Auf diese Weise könnten Sie sämtliche Werte eines Schlüssels auflisten, die ein bestimmtes Stichwort enthalten. Die nächste Zeile listet alle Werte auf, die das Wort »Proxy« enthalten:

```
PS > $key = Get-Item $pfad
PS > $key.GetValueNames() | Where-Object { $_ -like '*proxy*' } | ForEach-Object →
    { '{0,10} = {1}' -f $_, $key.GetValue($_) }
AutoConfigProxy = wininet.dll
MigrateProxy = 1
ProxyEnable = 0
ProxyServer = proxy-server.bv.aok.de:80
ProxyOverride = <local>;*.local
```

Standardwert eines Schlüssels lesen

Sie möchten den Standardwert eines Schlüssels lesen. Der Standardwert wird im Registrierungs-Editor unter dem Namen (*Standard*) in der rechten Spalte angezeigt.

Lösung

Sprechen Sie den Standardwert unter dem besonderen Namen '*(default)*' an. Die folgende Zeile liest den Standardwert eines Schlüssels:

```
PS > $pfad = 'HKLM:\Software\Classes\.bmp'  
PS > (Get-ItemProperty $pfad).'(default)'  
Paint.Picture
```

Oder sprechen Sie den Registrierungsschlüssel an und rufen Sie die Methode *GetValue()* mit einem leeren String auf:

```
PS > $key = Get-Item 'HKLM:\Software\Classes\.bmp'  
PS > $key.GetValue('')  
Paint.Picture
```

Hintergrund

Der Standardwert muss nicht vorhanden sein. Ist er nicht definiert, wird »nichts« zurückgeliefert. Wollen Sie den Wert eines Schlüssels lieber mit der .NET-Methode *GetValue()* erfragen, beschaffen Sie sich zuerst mit *Get-Item* den Schlüssel. Danach haben Sie über den Punkt Zugriff auf seine Eigenschaften und Methoden: Geben Sie hinter den runden Klammern einen Punkt an, dürfen Sie also jetzt die Eigenschaften und Methoden verwenden, die das Objekt unterstützt.

Werte vieler Schlüssel auslesen

Sie möchten von sämtlichen Unterschlüsseln bestimmte Werte auslesen. Sie wollen zum Beispiel erfahren, welche Software auf einem Computer installiert ist.

Lösung

Verwenden Sie *Get-ItemProperty* und setzen Sie Platzhalterzeichen ein. Die folgende Zeile liest aus sämtlichen Unterschlüsseln von *HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall* eine Reihe von Werten aus:

```
PS > Get-ItemProperty HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\* →
    | Select-Object DisplayName, DisplayVersion, UninstallString
DisplayName      DisplayVersion      UninstallString
-----
Microsoft .NET Framework... 4.0.30319           MsiExec.exe...
Microsoft Office Office 64... 12.0.6425.1000      MsiExec.exe...
Microsoft Office Shared 64... 12.0.6425.1000      MsiExec.exe...
Microsoft Application Erro... 12.0.6015.5000
Apple Mobile Device Suppor... 3.3.0.69           MsiExec.exe...
(...)
```

Hintergrund

Geben Sie ein Platzhalterzeichen am Ende des Pfadnamens an, liest *Get-ItemProperty* die Werte aus sämtlichen Unterschlüsseln des genannten Registrierungsschlüssels. Mit *Select-Object* legen Sie dann fest, welche Werte Sie sehen möchten.

Dabei kann es vorkommen, dass einzelne Spalten leer bleiben. Das geschieht, wenn im jeweiligen Schlüssel dieser Wert nicht vorkommt. Um solche leeren Einträge aus der Liste auszufiltern, greifen Sie zu *Where-Object*:

```
PS > Get-ItemProperty →
    HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\* | →
    Select-Object DisplayName, DisplayVersion, UninstallString | →
    Where-Object { $_.DisplayName -ne $null }
```

Prüfen, ob ein bestimmter Wert existiert

Sie möchten erfahren, ob in einem Registrierungsschlüssel ein bestimmter Wert vorhanden ist.

Lösung

Greifen Sie auf den Registrierungsschlüssel zu und erfragen Sie die Liste der darin vorhandenen Werte. Mit dem Operator *-contains* prüfen Sie, ob der gesuchte Wert in der Liste vorkommt:

```
PS > @((Get-Item HKCU:\Software\Testschlüssel).Property) -contains "Wert1"
False
```

Hintergrund

Jeder Registrierungsschlüssel enthält die Eigenschaft *Property*, die eine Liste mit den Namen der Werte liefert, die in diesem Schlüssel vorhanden sind. Um auf die Eigenschaft *Property* zugreifen zu können, beschaffen Sie sich zuerst mit *Get-Item* den Registrierungsschlüssel und stellen diese Anweisung in runde Klammern, damit sie von PowerShell zuerst ausgewertet wird.

Weil es sein kann, dass ein Schlüssel nur einen oder überhaupt keinen Wert enthält, wandeln Sie das Ergebnis von *Property* explizit in ein Feld um. Dazu betten Sie die Anweisung ein in *@()*. Selbst wenn der Schlüssel nur einen oder gar keinen Wert enthält, erhalten Sie so ein Feld mit einem oder gar keinem Element.

Dieses Feld kann jetzt vom Operator *-contains* ausgewertet werden. Der Operator liefert *\$true* zurück, wenn das Suchwort in einem Feldelement vorkommt, sonst *\$false*.

Wert eines Schlüssels ändern

Sie möchten einem vorhandenen Registrierungsschlüssel einen neuen Wert hinzufügen oder einen vorhandenen Wert ändern.

Lösung

Ändern Sie Werte mit *Set-ItemProperty*. Existiert der Wert noch nicht, wird er angelegt:

```
PS > $key = 'HKCU:\Software\Testschlüssel'
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
    Out-Null; 'Testschlüssel angelegt' }
PS >
PS > Set-ItemProperty $key Texteintrag -value "Der Inhalt des Wertes" -type String
PS > Set-ItemProperty $key '(default)' -value "Legt den Standardwert fest" →
    -type String
PS > Set-ItemProperty $key 'Ein Binärwert' -value (1,2,3,4) -type Binary
PS > Set-ItemProperty $key Windows-Ordner -value "%WINDIR%" -type ExpandString
PS > Get-ItemProperty $key
```

```
PSPath :
Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Testschlüssel
PSParentPath : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software
PSChildName : Testschlüssel
PSDrive : HKCU
PSProvider : Microsoft.PowerShell.Core\Registry
(default) : Legt den Standardwert fest
Texteintrag : Der Inhalt des Wertes
Ein Binärwert : {1, 2, 3, 4}
Windows-Ordner : C:\Windows
```

Hintergrund

Set-ItemProperty überschreibt den Inhalt eines Werts. Falls der Wert noch nicht existiert, wird er angelegt. Damit ist *Set-ItemProperty* der unproblematischste Weg, schnell und einfach einem Registrierungsschlüssel Werte zuzuweisen.

Wenn Sie verhindern möchten, dass schon vorhandene Werte durch neue Werte überschrieben werden, setzen Sie anstelle von *Set-ItemProperty* das Cmdlet *New-ItemProperty* ein. *New-Item-*

Property fügt neue Werte hinzu, kann aber bereits vorhandene Werte nicht überschreiben (es sei denn, Sie geben zusätzlich den Parameter *-force* an):

```
PS > $key = 'HKCU:\Software\Testschlüssel'
PS > $wertname = 'Mein Eintrag'
PS > $wertinhalt = 'Neuer Text für den Eintrag'
PS >
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
    Out-Null; 'Testschlüssel angelegt' }
Testschlüssel angelegt
PS >
PS > New-ItemProperty $key $wertname -value $wertinhalt -ea →
    SilentlyContinue > $null
PS > if (!$?) {
>> "Der Wert '$wertname' existiert bereits und kann nicht überschrieben werden."
>> } else {
>> "Der Wert '$wertname' wurde geschrieben."
>> }
>>
Der Wert 'Mein Eintrag' wurde geschrieben.
PS > New-ItemProperty $key $wertname -value $wertinhalt -ea →
    SilentlyContinue > $null
PS > if (!$?) {
>> "Der Wert '$wertname' existiert bereits und kann nicht überschrieben werden."
>> } else {
>> "Der Wert '$wertname' wurde geschrieben."
>> }
>>
Der Wert 'Mein Eintrag' existiert bereits und kann nicht überschrieben werden.
```

In diesem Beispiel wird *New-ItemProperty* mit der *ErrorAction SilentlyContinue* aufgerufen, sodass Fehlermeldungen verschluckt werden. Das Ergebnis von *New-ItemProperty* wird an *\$null* weitergeleitet, denn im Gegensatz zu *Set-ItemProperty* liefert *New-ItemProperty* als Ergebnis im Erfolgsfall den angelegten Wert zurück, der hier nicht weiter benötigt wird.

Ob *New-ItemProperty* erfolgreich war oder nicht, verrät anschließend die Variable *\$?*. Sie enthält *\$false*, wenn der vorangegangene Befehl einen Fehler verursachte. Mit dem Nicht-Operator *-not* (Kurzform *!*) kann dann darauf reagiert werden.

Die Werte eines Schlüssels können auch mithilfe von Methoden des Registrierungsschlüssels gelesen und verändert werden. Veränderungen sind allerdings auf diese Weise nur möglich, wenn der Schlüssel mit Schreibrechten geöffnet wurde. *Get-Item* öffnet Registrierungsschlüssel allerdings grundsätzlich nur mit Leserechten. Sie müssten in diesem Fall also manuell zur entsprechenden .NET-Methode *OpenSubKey()* greifen, um den Schlüssel tatsächlich mit Schreibrechten zu öffnen:

```

PS > $key = 'HKCU:\Software\Testschlüssel'
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
    Out-Null; 'Testschlüssel angelegt' }
PS >
PS > # Erster Ansatz schlägt fehl, denn es fehlen Schreibrechte:
PS > $regkey = Get-Item $key
PS > $regkey.SetValue('Wert 5', 'Ein neuer Wert')
Ausnahme beim Aufrufen von "SetValue" mit 2 Argument(en): "In den Registrierungsschlüssel
kann nicht geschrieben werden."
Bei Zeile:1 Zeichen:17
+ $regkey.SetValue( <<<< 'Wert 5', 'Ein neuer Wert')
PS >
PS > # Zweiter Ansatz funktioniert, weil der Schlüssel mit Schreibrechten →
    geöffnet wurde:
PS > $regkey = →
    [Microsoft.Win32.Registry]::CurrentUser.OpenSubKey($key.SubString(6), $true)
PS > $regkey.SetValue('Wert 5', 'Ein neuer Wert')
PS > $regkey.Close()

```

WICHTIG

Beachten Sie, dass die Methode *OpenSubKey()* immer an einen bestimmten Registrierungszweig gebunden ist. Deshalb muss der erste Teil des Registrierungspaths, im Beispiel also »HKCU:\«, entfernt werden. Das erledigt die Methode *SubString()*, die den Inhalt von *\$key* erst ab Zeichenposition 6 ausgibt.

Über .NET Framework steht Ihnen außerdem die statische Methode *SetValue()* zur Verfügung, mit der Sie Werte eines Schlüssels direkt hinzufügen und ändern können:

```

PS > [Microsoft.Win32.Registry]::SetValue→
    ('HKEY_CURRENT_USER\Software\Testschlüssel', 'Neuer Wert', 1000)

```

WICHTIG

Die Methode *SetValue()* erwartet einen vollqualifizierten Pfadnamen zum Registrierungsschlüssel, dem Sie einen Wert hinzufügen wollen. Verwenden Sie hier also nicht die virtuellen PowerShell-Laufwerke *HKLM:* oder *HKCU:*, sondern schreiben Sie die Wurzelnamen der Registrierung jeweils aus: »HKEY_LOCAL_MACHINE«, bzw. »HKEY_CURRENT_USER«.

Wert eines Schlüssels löschen

Sie möchten den Inhalt eines Werts oder den ganzen Wert mitsamt seinem Inhalt löschen.

Lösung

Verwenden Sie *Clear-ItemProperty*, wenn Sie nur den Inhalt des Werts löschen wollen. Der Wert bleibt erhalten:

```
PS > $key = 'HKCU:\Software\Testschlüssel'
PS >
PS > # Schlüssel und Wert zuerst testweise anlegen:
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
    Out-Null; 'Testschlüssel angelegt' }
PS > Set-ItemProperty $key Testwert -value Testinhalt
PS >
PS > Clear-ItemProperty $key Testwert
```

Möchten Sie den Wert insgesamt entfernen, verwenden Sie *Remove-ItemProperty*:

```
PS > Remove-ItemProperty $key Testwert
```

Hintergrund

Während *Clear-ItemProperty* nur den Inhalt eines Werts löscht, den Wert aber behält, entfernt *Remove-ItemProperty* den gesamten Wert samt Inhalt. Beide Cmdlets unterstützen Platzhalter, sodass Sie auch mehrere Werte auf einmal löschen oder entfernen können.

Im folgenden Beispiel wird ein Testschlüssel angelegt. Dem Schlüssel werden zehn Werte hinzugefügt. Diese Werte werden dann mit *Remove-ItemProperty* und einem Platzhalterzeichen wieder entfernt.

ACHTUNG Immer wenn Sie Platzhalterzeichen verwenden, sollten Sie zusätzlich die Parameter *-whatIf* beziehungsweise *-confirm* einsetzen, damit Sie die tatsächlich durchgeführten Aktionen prüfen und überwachen können. Andernfalls besteht die Gefahr, dass Ihre Platzhalterzeichen mehr Werte betreffen als vorgesehen.

```
PS > $key = 'HKCU:\Software\Testschlüssel'
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
    Out-Null; 'Testschlüssel angelegt' }
PS > 1..10 | % {New-ItemProperty $key "Wert$_" -value $_ -> $null }
PS > Remove-ItemProperty $key Wert* -whatIf
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert1".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert2".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert3".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert4".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert5".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert6".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert7".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert8".
WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert9".
```

```

WhatIf: Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert10".
PS > Remove-ItemProperty $key Wert* -confirm
Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Eigenschaft entfernen" für das Ziel "Element:
HKEY_CURRENT_USER\Software\Testschlüssel Eigenschaft: Wert1".
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe (Der Standardwert
ist "J"): j

```

Standardwert eines Schlüssels löschen

Sie möchten den Standardwert eines Registrierungsschlüssels löschen. Der Standardwert wird im Registrierungs-Editor in der rechten Spalte unter dem Namen (*Standard*) angezeigt und ist häufig anfangs undefiniert.

Lösung

Der Standardwert eines Registrierungsschlüssels kann nur über .NET Framework entfernt werden, wenn er einmal festgelegt wurde:

```

PS > [Microsoft.Win32.Registry]::CurrentUser.OpenSubKey('Software\Testschlüssel' →
, $true).DeleteValue("")

```

Sie können den Standardwert natürlich nur löschen, wenn er vorher angelegt wurde:

```

PS > Set-ItemProperty HKCU:\Software\Testschlüssel '(default)' 'Ein Standardwert'

```

Hintergrund

Eine Sonderstellung nimmt der Standardwert eines Schlüssels ein. Anfangs ist dieser Wert undefiniert. Sobald Sie diesen Wert festlegen, verhält er sich anders als alle benannten Werte. Sie können seinen Inhalt zwar mit *Clear-ItemProperty* löschen, den Wert selbst aber nicht mit *Remove-ItemProperty* entfernen:

```

PS > $key = 'HKCU:\Software\Testschlüssel'
PS > If (!(Test-Path $key)) { md HKCU:\Software\Testschlüssel | →
Out-Null; 'Testschlüssel angelegt' }
PS >
PS > Set-ItemProperty $key '(default)' -value 'Neuer Wert'
PS > Clear-ItemProperty $key '(default)'
PS >
PS > Remove-ItemProperty $key '(default)'
Remove-ItemProperty : Die Eigenschaft (default) ist im Pfad
HKEY_CURRENT_USER\Software\Testschlüssel nicht vorhanden.
Bei Zeile:1 Zeichen:20
+ Remove-ItemProperty <<<< $key '(default)'

```

Um den Standardwert zu entfernen, müssen Sie daher die Basisfunktionen von .NET Framework direkt nutzen und den Schlüssel mit Schreibrechten öffnen. Danach können Sie den Standardwert unter Angabe eines leeren Texts als Wertname mit *DeleteValue()* löschen:

```
PS > [Microsoft.Win32.Registry]::CurrentUser.OpenSubKey('Software\Testschlüssel' →  
    , $true).DeleteValue("")
```

Windows-Registrierungs-Editor öffnen

Sie möchten die Ergebnisse Ihrer PowerShell-Anweisungen im Registrierungs-Editor kontrollieren.

Lösung

Öffnen Sie den Registrierungs-Editor:

```
PS > regedit
```

Hintergrund

Weil der Registrierungs-Editor *regedit.exe* im *Windows*-Ordner gespeichert ist und dieser in der Umgebungsvariablen *\$env:Path* geführt wird, können Sie dieses Programm ohne Angabe eines qualifizierten Pfadnamens und ohne Dateierweiterung einfach über »regedit« starten.

```
PS > Get-Command regedit
```

CommandType	Name	Definition
-----	----	-----
Application	regedit.exe	C:\Windows\regedit.exe

regedit ist ein Single Instance-Programm, kann also nur einmal geöffnet werden. Starten Sie es erneut, erhalten Sie keine zweite Instanz des Programms. Es ist gut geeignet, die Änderungen in der Registrierungsdatenbank zu kontrollieren. Allerdings wäre es wünschenswert, wenn man *regedit* unter Angabe eines bestimmten Registrierungszeigs so öffnen könnte, dass dieser Zweig sofort angezeigt wird.

Die folgende Funktion *Regedit* fügt diese Möglichkeit hinzu. Rufen Sie *Regedit* auf, prüft die Funktion zunächst, ob *regedit.exe* bereits ausgeführt wird. Falls ja, wird der Start abgebrochen.

Geben Sie als Parameter einen Registrierungszeig an, wird *regedit.exe* so gestartet, dass dieser Zweig vorgewählt ist. Geben Sie keinen Parameter an und liegt das aktuelle Verzeichnis in der Registrierungsdatenbank, wird das aktuelle Verzeichnis im Registrierungs-Editor angezeigt.

```

Function Regedit ([string]$path = $(Get-Location)) {
    if (@(Get-Process regedit -ea SilentlyContinue).Count -gt 0) {
        Throw "REGEDIT wird bereits ausgeführt."
    }

    $key = Get-Item $path
    if ($key.GetType().Name -eq 'RegistryKey') {
        $regpath = $key.name
        $key = →
        "HKCU:\Software\Microsoft\Windows\CurrentVersion\Applets\Regedit"
        Set-ItemProperty $key LastKey "Computer\$regpath"
    }
    regedit.exe
}

PS > # Registrierungs-Editor mit einem vorgewählten Zweig öffnen:
PS > regedit HKCU:\Software
PS >
PS > # Registrierungs-Editor mit aktuellem Pfad öffnen:
PS > cd HKLM:\System
PS > regedit

```

Die Funktion verwendet als Standardvorgabe für den Parameter den aktuellen Pfad, den *Get-Location* liefert. Geben Sie also keinen eigenen Pfad an, wird der aktuelle Pfad verwendet.

Die Funktion greift nun mit *Get-Item* auf den angegebenen Pfad zu und überprüft mit *GetType()*, ob es sich wirklich um einen Registrierungszweig handelt. In diesem Fall liefert die *Name*-Eigenschaft den Pfadnamen des Zweigs. Dieser wird an einer besonderen Stelle in der Registrierungsdatenbank hinterlegt. Anschließend wird *regedit.exe* gestartet.

In der Registrierungsdatenbank navigieren

Sie wollen die Registrierungsdatenbank öffnen und darin Informationen suchen.

Lösung

Bewegen Sie sich in der Registrierungsdatenbank genau wie im Dateisystem. PowerShell unterstützt die virtuellen Laufwerke *HKLM:* (für den Registrierungszweig *HKEY_LOCAL_MACHINE*) und *HKCU:* (für den Registrierungszweig *HKEY_CURRENT_USER*):

```

PS > cd HKCU:
PS > dir

    Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER

SKC  VC Name                                Property
---  -
  2   0 AppEvents                          {}
  2   37 Console                           {ColorTable00, ColorTable01,...
 14   0 Control Panel                       {}
  0   2 Environment                        {TEMP, TMP}

```

(...)

PS > **cd Software**

PS > **dir**

Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software

SKC	VC Name	Property
----	----	-----
3	0 Adobe	{}
2	0 Analog Devices	{}
1	0 AppDataLow	{}
1	0 Apple Computer, Inc.	{}

(...)

Sie können jederzeit zurück zu Ihrem Stammverzeichnis wechseln:

PS > **cd \$home**

Wenn Sie vor dem Wechsel in die Registrierung die aktuelle Position auf den Stapel legen, gelangen Sie nach einem Ausflug in die Registrierungsdatenbank auch wieder genau zu Ihrem Ausgangspunkt zurück:

```
PS > pushd
PS > cd HKLM:
PS > dir
PS > cd software
PS > dir
PS > popd
```

Registrierungszweige lassen sich auch ohne Wechsel des aktuellen Ordners anzeigen, indem Sie den kompletten Pfad angeben:

PS > **dir hklm:\software\microsoft**

Möchten Sie zwischenzeitlich lieber auf den Registrierungs-Editor von Windows zugreifen, starten Sie ihn folgendermaßen:

PS > **regedit**

Hintergrund

Da PowerShell einen speziellen Provider für die Registrierungsdatenbank mitbringt, können Sie in der Registrierungsdatenbank wie in einem Laufwerk navigieren. Anfangs sind nur zwei virtuelle Laufwerke eingerichtet, nämlich *HKLM:* und *HKCU:*

```
PS > (Get-PSProvider Registry).Drives
```

Name	Provider	Root	CurrentLocation
HKLM	Registry	HKEY_LOCAL_MACHINE	System
HKCU	Registry	HKEY_CURRENT_USER	Software

```
PS > Get-PSDrive -PSProvider Registry
```

Name	Provider	Root	CurrentLocation
HKCU	Registry	HKEY_CURRENT_USER	Software
HKLM	Registry	HKEY_LOCAL_MACHINE	System

Es steht Ihnen frei, mit *New-PSDrive* weitere Laufwerke hinzuzufügen. Diese Laufwerke könnten Abkürzungen sein für häufig benötigte Zweige innerhalb der vorhandenen Laufwerke. Neue Laufwerke können aber auch ganz andere Zweige der Registrierungsdatenbank zugänglich machen.

```
PS > New-PSDrive HKCR Registry HKEY_CLASSES_ROOT
PS > Dir HKCR:
```

Andere Orte der Registrierungsdatenbank ansprechen

Sie möchten mehr Registrierungszweige ansprechen als die vordefinierten Zweige *HKLM* und *HKCU*.

Lösung

Fügen Sie entweder weitere virtuelle Laufwerke hinzu. Um den Zweig *HKEY_CLASSES_ROOT* unter dem virtuellen Laufwerk *HKCR*: ansprechen zu können, verwenden Sie *New-PSDrive*:

```
PS > New-PSDrive HKCR Registry HKEY_CLASSES_ROOT
```

Name	Provider	Root	CurrentLocation
HKCR	Registry	HKEY_CLASSES_ROOT	

```
PS > dir HKCR:
```

```
Hive: Microsoft.PowerShell.Core\Registry::HKEY_CLASSES_ROOT
```

SKC	VC	Name	Property
3	10	*	{AlwaysShowExt, PreviewDetails, PreviewT...}
1	2	.386	{(default), PerceivedType}
1	2	.3g2	{(default), Content Type}
1	2	.3gp	{(default), Content Type}
1	2	.3gp2	{(default), Content Type}
1	2	.3gpp	{(default), Content Type}
(...)			

Oder Sie sprechen den gewünschten Registrierungszweig unter seinem regulären Namen an und fügen am Anfang dieses Pfadnamens den Providernamen an: »Registry::«:

```
PS > Dir Registry::HKEY_CLASSES_ROOT
```

Hintergrund

Mit *New-PSDrive* legen Sie neue virtuelle Laufwerke an und müssen dazu drei Informationen liefern: den Namen des neuen Laufwerks, den Provider für den Informationsspeicher und das Wurzelverzeichnis des neuen Laufwerks. Das neue Laufwerk bleibt in der gesamten Sitzung erhalten, ist allerdings nur in der PowerShell-Konsole verwendbar, in der es angelegt wurde. Nachdem Sie die Konsole schließen und neu öffnen, ist das Laufwerk wieder verschwunden. Möchten Sie es dauerhaft einrichten, fügen Sie den Befehl zum Anlegen des Laufwerks in Ihr Autostart-Skript ein, dessen Pfadname Sie in der Variablen *\$profile* erfahren.

Sie brauchen allerdings gar keine neuen virtuellen Laufwerke anzulegen, wenn es Ihnen nur darum geht, aus anderen Orten der Registrierungsdatenbank Informationen zu lesen. In diesem Fall fügen Sie vor dem Pfadnamen des gewünschten Registrierungsschlüssels den Namen des zuständigen Providers an: »Registry::«. Weil PowerShell nun weiß, welcher Provider den angegebenen Pfadnamen korrekt interpretieren kann, lassen sich die Informationen auf diese Weise auch ohne virtuelle Laufwerke abrufen.

Remote auf Registrierungsdatenbank zugreifen

Sie möchten Informationen aus der Registrierungsdatenbank eines fremden Computers lesen oder ändern.

Lösung

Verwenden Sie entweder die .NET-Methoden, um über das Netzwerk auf die Registrierungsdatenbank eines anderen Computers zuzugreifen:

```
$remote = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey("LocalMachine", →  
"192.168.2.108")  
$key = $remote.OpenSubKey("Software\Microsoft\Windows\CurrentVersion")  
Foreach ($subkeyname in $key.GetSubKeyNames() ) {  
$subkey = $key.OpenSubKey($subkeyname)  
$rv = [PSObject] $subKey  
$rv | Add-Member NoteProperty PSChildName $subkeyname  
$rv | Add-Member NoteProperty Property $subkey.GetValueNames()  
$rv  
$subkey.close()  
}
```

Oder setzen Sie das universelle PowerShell-Remoting ein, wenn auf dem Zielcomputer ebenfalls PowerShell 2.0 ausgeführt wird und dort das Remoting eingeschaltet wurde:

```
PS > Invoke-Command { Dir HKLM:\Software } -ComputerName PC01 -Credential →
Administrator
```

Hintergrund

Der Registry-Provider von PowerShell kann nicht remote auf eine fremde Registrierungsdatenbank zugreifen, die zugrunde liegende .NET Framework-Version schon: *OpenRemoteBaseKey()*. Um eine fremde Registrierungsdatenbank damit anzusprechen, sind allerdings eine ganze Reihe von Voraussetzungen zu erfüllen:

- Sie benötigen Zugriffsberechtigungen auf das Zielsystem
- Zwischen Ihnen und dem Remotesystem dürfen keine Firewalls vorhanden sein, die RPC-Verbindungen behindern
- Auf beiden Systemen muss der Remoteregistrierungsdienst ausgeführt werden

Damit Sie die Ergebnisse, die *OpenRemoteBaseKey()* liefert, in genau derselben komfortablen Weise nutzen können wie die Ergebnisse des Registry-Providers für die lokale Registrierungsdatenbank, wandeln Sie die Ergebnisobjekte um: Mit *[PSObject]* sorgen Sie dafür, dass die .NET-Objekte in den Typ *PSObject* umgewandelt werden. Für vollständige Kompatibilität müssen dann noch die Eigenschaften *Property* und *PSChildName* mit *Add-Member* jedem Objekt hinzugefügt werden.

Alternativ können Sie auch PowerShell 2.0 Remoting einsetzen, um auf ein anderes System zuzugreifen. Dazu muss Remoting aktiviert sein und das Zielsystem ebenfalls PowerShell 2.0 verwenden.

HINWEIS

Denken Sie daran, dass Sie bei allen Remotetechniken unter Ihrem Benutzernamen (oder bei Verwendung des Parameters *-Credential* unter dem Namen des angegebenen Benutzers) angemeldet werden. Sie können remote den allgemeinen Zweig *HKEY_LOCAL_MACHINE* auslesen. Der Zweig *HKEY_CURRENT_USER* repräsentiert hingegen immer den Anwender, unter dessen Namen Sie sich angemeldet haben. Möchten Sie auf die Informationen für andere Benutzer zugreifen, schauen Sie im Zweig *HKEY_USERS*, ob der benötigte Registrierungszweig geladen ist. Die nächste Zeile verwendet dafür das PowerShell-Remoting:

```
PS > Invoke-Command { Dir Registry::HKEY_USERS } -ComputerName storage1
```

```
Hive: HKEY_USERS
```

SKC	VC Name	Property	PSComputerName
---	----	-----	-----
10	0 .DEFAULT	{}	storage1
10	0 S-1-5-19	{}	storage1
2	0 S-1-5-19_Classes	{}	storage1
10	0 S-1-5-20	{}	storage1
2	0 S-1-5-20_Classes	{}	storage1
8	0 S-1-5-21-3190518249-4009757...	{}	storage1
0	0 S-1-5-21-3190518249-4009757...	{}	storage1
10	0 S-1-5-18	{}	storage1

Ist der gewünschte Zweig noch nicht geladen, können Sie diesen mit dem Befehl *reg.exe* vorübergehend aus der Datei *ntuser.dat* des Benutzers nachladen:

```
PS > reg.exe /load HKU\Testuser c:\users\wilhelm\ntuser.dat
PS > dir Registry::HKEY_USERS\Testuser\Software
```

Zusammenfassung

Die Verwaltung der Registrierungsschlüssel verläuft ganz analog zu Dateien und Ordnern im Dateisystem und Sie dürfen hierfür dieselben Cmdlets einsetzen, die Sie auch schon im letzten Kapitel verwendet haben. Für PowerShell sind Registrierungsschlüssel bloß Items, genau wie Dateien und Ordner.

Die Werte eines Registrierungsschlüssels dagegen werden etwas anders verwaltet. Sie bezeichnet PowerShell als *ItemProperties*, die zu einem Item gehören. Greifen Sie also zu den Cmdlets aus der Familie *ItemProperty*, um Registrierungswerte zu lesen, zu ändern, anzulegen oder zu entfernen:

```
PS > Get-Command -noun ItemProperty
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-ItemProperty	Clear-ItemProperty [-Path] <String[]>...
Cmdlet	Copy-ItemProperty	Copy-ItemProperty [-Path] <String[]>...
Cmdlet	Get-ItemProperty	Get-ItemProperty [-Path] <String[]> ...
Cmdlet	Move-ItemProperty	Move-ItemProperty [-Path] <String[]>...
Cmdlet	New-ItemProperty	New-ItemProperty [-Path] <String[]> ...
Cmdlet	Remove-ItemProperty	Remove-ItemProperty [-Path] <String[...>
Cmdlet	Rename-ItemProperty	Rename-ItemProperty [-Path] <String>...
Cmdlet	Set-ItemProperty	Set-ItemProperty [-Path] <String[]> ...

Als Vorgabe legt PowerShell virtuelle Laufwerke lediglich für die Haupthives *HKEY_LOCAL_MACHINE* und *HKEY_CURRENT_USER* an. Sie können aber mit *New-PSDrive* weitere Laufwerke hinzufügen, die auch die übrigen Hives (oder andere Orte in der Registry) repräsentieren. Noch einfacher: Verwenden Sie den regulären Pfadnamen des Hives oder Schlüssels und stellen Sie lediglich den Providernamen voran, also *Registry::HKEY_USERS*. Auf diese Weise brauchen Sie überhaupt keinen Laufwerksnamen zu nutzen.

Kapitel 9

Prozesse und Anwendungen

In diesem Kapitel:

Laufende Prozesse sichtbar machen	294
Feststellen, ob ein Prozess läuft	297
Anzahl der Instanzen eines Prozesses bestimmen	298
Prozess starten	299
Prozess unter anderer Identität starten	301
Prozess als Administrator starten	304
Prozess beenden	305
Abgestürzte Prozesse finden und beenden	307
Abkürzungen für häufige Befehle einrichten	307
Ausgaben eines Programms weiterverarbeiten	309
Zusammenfassung	312

Die Verwaltung laufender Anwendungen gehört zu den Routineaufgaben eines Administrators. PowerShell unterstützt Sie dabei, indem Sie nicht nur laufende Prozesse auflisten, sondern auch analysieren können, um beispielsweise nicht mehr reagierende Anwendungen zu finden, die Prozessorbelastung zu ermitteln oder die Laufzeit eines Prozesses zu bestimmen.

Darüber hinaus kann PowerShell natürlich auch neue Prozesse starten, wobei Sie dank umfangreicher Optionen die Möglichkeit haben, Prozesse auch unter dem Namen eines anderen Benutzers oder mit höherer (oder niedrigerer) Priorität auszuführen.

Laufende Prozesse sichtbar machen

Sie möchten sehen, welche Prozesse auf Ihrem Computer augenblicklich ausgeführt werden.

Lösung

Verwenden Sie *Get-Process*. Das Cmdlet liefert eine Liste der aktuell ausgeführten Prozesse:

PS > **Get-Process**

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
27	1	364	1252	11		384	AEADISRV
212	6	13544	11900	47		1300	audiodg
35	2	892	3748	40		2432	conime
672	8	2120	3124	73		644	csrss
(...)							

Hintergrund

Get-Process ruft eine Liste sämtlicher ausgeführter Prozesse ab. Nur wenn Sie über Administratorrechte verfügen, sehen Sie alle Informationen sämtlicher Prozesse. Ohne Administratorrechte erhalten Sie nur für Ihre eigenen Prozesse volle Informationen. Prozesse, die im Namen anderer Benutzer laufen, liefern keine im weitesten Sinne personenbezogenen Daten wie beispielsweise die CPU-Last oder den zugrunde liegenden Programmpfad.

Jeder laufende Prozess entspricht einem *Process*-Objekt, das alle wesentlichen Informationen über den Prozess enthält. PowerShell zeigt zunächst nur die wichtigsten Prozessinformationen an. Wenn Sie jedoch die Ergebnisse an *Select-Object* weiterreichen und explizit sämtliche Objekteigenschaften anzeigen lassen, erhalten Sie viel mehr Informationen:

PS > **Get-Process**

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
27	1	364	1252	11		384	AEADISRV
212	6	13544	11736	47		1300	audiodg
35	2	892	3748	40		2432	conime
(...)							

```
PS > Get-Process | Select-Object *
NounName      : Process
Name           : AEADISRV
Handles       : 27
VM             : 11948032
WS            : 1282048
PM            : 372736
NPM           : 960
Path          :
Company       :
CPU           :
FileVersion   :
ProductVersion :
Description    :
Product       :
Id            : 384
PriorityClass  :
(...)
```

Interessieren Sie sich nur für einen bestimmten Prozess, geben Sie den Namen des gewünschten Prozesses an. Die folgende Zeile listet sämtliche Instanzen des Prozesses *explorer* auf.

```
PS > Get-Process explorer

Handles  NPM(K)    PM(K)      WS(K) VM(M)    CPU(s)    Id ProcessName
-----
1328     59        146268     67048  402      628,28    4092 explorer
```

Möchten Sie die Anzahl ausgeführter Prozesse ermitteln, greifen Sie entweder zu *Measure-Object* oder Sie speichern das Ergebnis als Array und rufen dessen Eigenschaft *Count* ab. Beide Ansätze melden in diesem Fall die Anzahl der laufenden PowerShell-Prozesse.

```
PS > Get-Process powershell -ErrorAction SilentlyContinue | Measure-Object

Count      : 1
Average    :
Sum        :
Maximum    :
Minimum    :
Property   :

PS > Get-Process powershell -ErrorAction SilentlyContinue | Measure-Object | →
Select-Object -expand Count
1
PS > @(Get-Process powershell -ErrorAction SilentlyContinue).Count
1
```

Lassen Sie dabei Fehlermeldungen mit `-ErrorAction SilentlyContinue` unterdrücken, denn falls der gesuchte Prozess überhaupt nicht ausgeführt würde, käme es sonst zu einer störenden Fehlermeldung.

Auch Platzhalterzeichen sind erlaubt. Die folgende Zeile listet alle Prozesse auf, deren Name mit »a« beginnt:

```
PS > Get-Process a*
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
27	1	364	1252	11		384	AEADISRV
215	6	13552	11752	47		1300	audiodg

Mit dem Filter *Where-Object* (Kurzform »?*«*) lassen sich beliebige Eigenschaften des *Process*-Objekts als Filterkriterium verwenden. Die nächste Zeile listet alle Prozesse auf, die mehr als 50 Mbyte Speicher verwenden:

```
PS > Get-Process | Where-Object { $_.Workingset -gt 50MB } | Sort-Object workingset
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
257	10	83520	53616	158		7876	mfpmp
1331	59	146344	66556	402	628,72	4092	explorer
820	33	142156	70108	398	140,45	8308	PowerShellPlus
755	19	85616	77828	188		1140	svchost
584	21	173132	86592	378	239,66	9444	Tunebite
3344	55	101272	127384	490	16,91	6068	OUTLOOK
1081	43	72144	127996	475	35,19	9920	WINWORD

Möchten Sie Prozesse sehen, die in den letzten zehn Minuten gestartet wurden, greifen Sie auf die Eigenschaft *StartTime* zurück und berechnen daraus die bisherige Laufzeit des Prozesses:

```
PS> Get-Process | Where-Object { try { (New-Timespan $_.StartTime).TotalMinutes -le 10 } catch { $false } }
PS> Get-Process | Where-Object { trap { Continue } (New-Timespan $_.StartTime).TotalMinutes -le 10 }
```

Beide Zeilen liefern dasselbe Resultat zurück, verwenden aber unterschiedliche Methoden, um Fehler zu unterdrücken. Fehler können auftreten, wenn Sie nicht über Administratorberechtigungen verfügen, weil Sie in diesem Fall nur die Startzeiten Ihrer eigenen Prozesse lesen dürfen. Da Sie zum Filtern einen berechneten Wert verwenden, wäre es sinnvoll, diesen Wert in das Ergebnisobjekt mit aufzunehmen, damit Sie die Laufzeit der Prozesse auch sichtbar anzeigen können. Nehmen Sie deshalb die neu berechnete Eigenschaft mit *Add-Member* in das Objekt auf.


```
Get-Process | ForEach-Object {
    try {
        $laufzeit = [Int](New-Timespan $_.StartTime).TotalMinutes
    } catch {
        $laufzeit = 9999999
    }
    $_ | Add-Member NoteProperty 'Laufzeit' $laufzeit -PassThru
} | Where-Object {
    $_.Laufzeit -le 10
} | Sort-Object Laufzeit | Select-Object name, laufzeit, starttime
```

Name	Laufzeit	StartTime
----	-----	-----
notepad	1	12.01.2011 12:12:08
iexplore	5	12.01.2011 12:08:10
WTGU	8	12.01.2011 12:05:15
notepad	10	12.01.2011 12:03:31

Bei der Gruppierung geben Sie eine Eigenschaft an, die als Gruppierungskriterium dienen soll. Anschließend werden die Prozesse anhand dieser Eigenschaft in Gruppen eingeteilt. So erhalten Sie beispielsweise eine Liste der Hersteller der laufenden Prozesse:

```
PS > Get-Process | Group-Object Company
```

Count	Name	Group
-----	----	-----
1	Andrea Electronics Cor...	{AEADISRV}
5		{audiodg, Idle, mfpmp, Sys...}
51	Microsoft Corporation	{conime, csrss, csrss, dwm...}
4	Lenovo Group Limited	{cssauth, SUService, tvt_r...}
1	Lenovo Group Ltd.	{EZEJMNAP}
1	Lenovo	{ibmpmsvc}
1	Eset	{nod32krn}
1	Analog Devices, Inc.	{smax4pnp}
3	TechSmith Corporation	{SnagIt32, SnagPriv, TscHelp}
2	Synaptics, Inc.	{SynTPEnh, SynTPLpr}
1	IBM	{tvttcsd}
1	UPEK Inc.	{upeksvr}
1	Conexant Systems, Inc.	{XAudio}

Feststellen, ob ein Prozess läuft

Sie möchten wissen, ob ein bestimmter Prozess augenblicklich ausgeführt wird, zum Beispiel, um Doppelstarts zu verhindern.

Lösung

Suchen Sie mit *Get-Process* nach dem gewünschten Prozess. Wenn *Get-Process* dabei einen Fehler meldet und *\$null* zurückliefert, läuft der Prozess nicht. Die folgende Zeile prüft, ob der Windows-Editor (Notepad) ausgeführt wird:

```
PS > if ((Get-Process notepad -ea SilentlyContinue) -eq $null) →
    { 'Notepad läuft nicht' } else { 'Notepad läuft' }
Notepad läuft nicht
PS > notepad
PS > if ((Get-Process notepad -ea SilentlyContinue) -eq $null) →
    { 'Notepad läuft nicht' } else { 'Notepad läuft' }
Notepad läuft
```

Hintergrund

Kann *Get-Process* den angegebenen Prozess nicht finden, meldet es einen Fehler und liefert kein Ergebnis. Die Fehlermeldung kann mit dem Parameter *-errorAction* (kurz: *-ea*) und der Einstellung *SilentlyContinue* verschluckt werden. Der Wert für »kein Ergebnis« entspricht *\$null*. Wenn also *Get-Process* als Ergebnis *\$null* liefert, konnte der angegebene Prozess nicht gefunden werden.

Anzahl der Instanzen eines Prozesses bestimmen

Sie wollen herausfinden, wie oft ein bestimmter Prozess ausgeführt wird.

Lösung

Beauftragen Sie *Get-Process*, den gewünschten Prozess zu finden, und speichern Sie das Ergebnis mit »@()« explizit in einem Feld. Anschließend liefert die Anzahl der Feldelemente in der Feldeigenschaft *Count* die Anzahl der laufenden Instanzen:

```
PS > @(Get-Process explorer -ea 0).Count
1
PS > "Derzeit laufen {0} Instanzen von PowerShell." -f →
    @(Get-Process powershell -ea 0).Count
Derzeit laufen 1 Instanzen von PowerShell.
```

Hintergrund

Wenn Sie das Ergebnis von *Get-Process* mit »@()« in einem Feld speichern, können Sie über *Count* ermitteln, ob (und wenn ja, wie oft) ein Prozess gerade ausgeführt wird. Der Zusatz »@()« sorgt dafür, dass das Ergebnis immer in einem Feld (Array) gespeichert wird. Normalerweise liefert *Get-Process* nur dann ein Feld zurück, wenn es mehr als einen Prozess gefunden hat. Mit dem Parameter *-ErrorAction SilentlyContinue* (Kurzform *-ea 0*) unterdrücken Sie Fehlermeldungen, die auftreten können, wenn überhaupt keine Instanzen des Programms ausgeführt werden.


Alternativ können Sie auch *Measure-Object* einsetzen:

```
PS > Get-Process explorer -ea 0 | Measure-Object | →  
    Select-Object -expand Count  
1
```

Prozess starten

Sie möchten einen neuen Prozess (eine neue Anwendung) starten.

Lösung

Im einfachsten Fall geben Sie den Namen des Prozesses ein und drücken die -Taste. Handelt es sich um einen konsolenbasierten Prozess, wird PowerShell unterbrochen, bis der gestartete Prozess beendet wird. Bei fensterbasierten Anwendungen setzt PowerShell sofort seine Arbeit fort.

```
PS > notepad
```

Programme, die nicht in einem Ordner lagern, der in der Umgebungsvariablen *Path* genannt wird, müssen mit einem vollständigen absoluten oder relativen Pfadnamen gestartet werden.

```
PS > $env:path  
C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Program Files (x86)\Common  
Files\MicrosoftShared\Windows  
Live;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsP  
owerShell\v1.0\;C:\Program Files (x86)\Windows Live\Shared;C:\Program Files  
(x86)\QuickTime\QTSystem\
```

Den Internet Explorer starten Sie deshalb zum Beispiel so:

```
PS > . "$env:programfiles\Internet Explorer\iexplore.exe"
```

Handelt es sich um ein konsolenbasiertes Programm, wird die Ausführung von PowerShell so lange unterbrochen, bis die konsolenbasierte Anwendung wieder beendet ist:

```
PS > ping.exe 10.10.10.10
```

Hintergrund

Wenn Sie einen Programmnamen eingeben, schaut PowerShell zunächst, ob dieser Programmname in einem der »vertrauenswürdigen« Ordner zu finden ist. Vertrauenswürdige Ordner sind alle Ordner, die in der *%path%*-Umgebungsvariable genannt werden. Der *Windows*-Ordner gehört zum Beispiel dazu, sodass sich der Notepad-Editor auch ohne Pfadangabe starten lässt.

Anders ist das bei *iexplore.exe*, dem Internet Explorer. Rufen Sie diesen analog wie Notepad auf, erhalten Sie eine Fehlermeldung:

```
PS > iexplore.exe
```

Die Benennung "iexplore.exe" wurde nicht als Cmdlet, Funktion, ausführbares Programm oder Skriptdatei erkannt. Überprüfen Sie die Benennung, und versuchen Sie es erneut.

Bei Zeile:1 Zeichen:12



```
+ iexplore.exe <<<<
```

Diese Anwendung können Sie nur starten, wenn Sie den absoluten oder relativen Pfadnamen angeben. Häufig lässt sich dieser Pfadname automatisiert zusammensetzen. Der Internet Explorer befindet sich beispielsweise immer im *Programme*-Ordner, den die Umgebungsvariable *programfiles* verrät. Er befindet sich darin im Unterordner *Internet Explorer*.

TIPP

Start-Process verfügt über etwas mehr Intelligenz. Mit diesem Cmdlet können Sie auch viele Windows-Programme starten, die nicht in einem der Ordner lagern, die die Umgebungsvariable *Path* aufführt:


```
PS > Start-Process iexplore
```

Tatsächlich verhält sich *Start-Process* genauso wie das *Ausführen*-Dialogfeld, das Sie mit  +  öffnen. Alle Programme, die Sie darin öffnen können, lassen sich auch mit *Start-Process* öffnen.

Weil der Pfadname Leerzeichen enthält, muss er in Anführungszeichen gestellt werden, und weil Sie in Ihrem Pfadnamen eine Umgebungsvariable auflösen wollen, muss der in doppelte und nicht etwa nur einfache Anführungszeichen gestellt werden:

```
PS > "$env:programfiles\Internet Explorer\iexplore.exe"
```

```
C:\Program Files\Internet Explorer\iexplore.exe
```

Durch die Anführungszeichen wird aus dem Pfadnamen ein Textstring, der sich nicht mehr über die -Taste ausführen lässt. Damit Ihr Text als ausführbare Anweisung verstanden wird, müssen Sie entweder ».« oder »&« davor schreiben und durch ein Leerzeichen vom Text trennen:

```
PS > . "$env:programfiles\Internet Explorer\iexplore.exe"
```

```
PS > & "$env:programfiles\Internet Explorer\iexplore.exe"
```

In diesem Beispiel funktionieren beide Zeilen gleich, aber wenn Sie auf diese Weise Skripts starten, gibt es einen Unterschied: Der Punkt sorgt dafür, dass alle Variablen im Skript global sind, während das kaufmännische Und-Zeichen den Prozess (also das Skript) isoliert in einem eigenen privaten Variablenbereich startet.

Sie können sich den Aufruf von Programmen über ihren langen Pfadnamen sparen, wenn Sie den Ordner, in dem das Programm enthalten ist, vertrauenswürdig machen – also einfach zur *Path*-Umgebungsvariablen hinzufügen:

```
PS > $env:path += ";$env:programfiles\Internet Explorer"
PS > iexplore
```

Diese Änderung der *Path*-Umgebungsvariablen gilt nur so lange, bis Sie PowerShell schließen. Sie können diese Anweisung aber in Ihr Profilskript aufnehmen, damit sie automatisch beim Start von PowerShell ausgeführt wird, oder aber die *Path*-Umgebungsvariable permanent ändern, zum Beispiel außerhalb von PowerShell.

Konsolenbasierte Anwendungen verhalten sich anders als fensterbasierte Anwendungen. Weil sie die Konsole für ihre Ausgaben benötigen, wird PowerShell vorübergehend angehalten, solange die konsolenbasierte Anwendung ausgeführt wird. In dieser Zeit nutzt dann die konsolenbasierte Anwendung die Konsole als Ein- und Ausgabeplattform. Sie können deshalb sehr bequem bei Bedarf zur klassischen Konsole wechseln und alte Konsolenbefehle eingeben. Sind sie fertig, verlassen Sie die alte Konsole mit dem Befehl *exit* wieder.

```
PS > cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Tobias>help
Geben Sie HELP "Befehlsname" ein, um weitere Informationen zu einem bestimmten
Befehl anzuzeigen.

ASSOC           Zeigt Dateierweiterungszuordnungen an bzw. ändert sie.
ATTRIB          Zeigt Dateiattribute an bzw. ändert sie.
BREAK           Schaltet die erweiterte Überprüfung für STRG+C ein bzw. aus.
BOOTCFG         Legt Eigenschaften zur Steuerung des Startladenvorganges in der
                Startdatenbank fest.
CACLS           Zeigt Datei-ACLs (Access Control List) an bzw. ändert sie.
(...)
VER             Zeigt die Windows-Version an.
VERIFY          Legt fest, ob das ordnungsgemäße Schreiben von Dateien auf
                den Datenträger überprüft werden soll.
VOL            Zeigt die Volumebezeichnung und die Seriennummer des
                Datenträgers an.
XCOPY           Kopiert Dateien und Verzeichnisstrukturen.
WMIC            Zeigt WMI-Informationen in der interaktiven Befehlsshell an.

Weitere Informationen finden Sie in der Befehlszeilenreferenz der Onlinehilfe.

C:\Users\Tobias>exit
```

Prozess unter anderer Identität starten

Sie möchten einen Prozess unter einem anderen Benutzernamen ausführen.

Lösung

Starten Sie den Prozess mit *Start-Process* und geben Sie mit dem Parameter *-Credential* den Namen des Benutzers ein, in dessen Name dieser Prozess ausgeführt werden soll. Sie werden

dann nach dem Kennwort gefragt und der Prozess wird im Namen des angegebenen Benutzers ausgeführt.

```
PS> Start-Process cmd.exe -Credential Tobias
```

Geben Sie den *Switch*-Parameter *-LoadUserProfile* mit an, falls das Benutzerprofil des Aufrufers geladen werden soll.

Möchten Sie dem Prozess zusätzliche Argumente mit übergeben, trennen Sie diese vom eigentlichen Programmaufruf und übergeben Sie die Argumente mit dem Parameter *-ArgumentList*.

Hintergrund

In PowerShell Version 2 wurde das Cmdlet *Start-Process* mit beinahe allen Parametern ausgestattet, die erforderlich sind, um Prozesse unter anderen Identitäten zu starten. Hinter den Kulissen greift *Start-Process* dabei auf die Methoden von .NET Framework zurück und verfährt ungefähr nach folgendem Muster:

```
PS > $anmeldung = Get-Credential
PS > $startinfo = New-Object System.Diagnostics.ProcessStartInfo
PS > $startinfo.UserName = $anmeldung.Username
PS > $startinfo.Password = $anmeldung.Password
PS > $startinfo.FileName = "$env:comspec"
PS > $startinfo.WorkingDirectory = "C:\"
PS > $startinfo.LoadUserProfile = $true
PS > $startinfo.UseShellExecute = $false
PS > [System.Diagnostics.Process]::Start($startinfo)
```

Möchten Sie die Anmeldeinformationen ohne Dialogfeld übergeben, erstellen Sie sich selbst das *Credential*-Objekt. Die folgenden Zeilen starten Notepad im Namen des Anwenders *contoso\Mueller*:

```
PS > $username = 'contoso\Mueller'
PS > $password = 'topSecret99'
PS > $cred = New-Object System.Management.Automation.PSCredential($username, →
($password | ConvertTo-SecureString -asPlainText -Force))
PS > Start-Process notepad -Credential $cred -LoadUserProfile
```

ACHTUNG Vermeiden Sie, sensible Informationen wie Kennwörter in Skripts zu nennen. Wenn dies unumgänglich ist, sollten Sie das Kennwort zumindest unleserlich machen. Das folgende Skript stellt die Funktion *Create-EncryptedCredentialScript* zur Verfügung:

```
function Create-EncryptedCredentialScript {
    param(
        [Parameter(Mandatory=$true)]
        $username,
        [Parameter(Mandatory=$true)]
        [System.Security.SecureString]
        $password,
        $filepath = "$env:temp\templatescript.ps1"
    )

    # Create empty template script:
    New-Item -ItemType File $filepath -Force -ErrorAction SilentlyContinue

    $key = 1..32 | ForEach-Object { Get-Random -Maximum 256 }
    $pwdencrypted = $password | ConvertFrom-SecureString -Key $key

    ('$password = "{0}"' -f $pwdencrypted) | Out-File $filepath
    ('$key = "{0}"' -f ($key -join ' ')) | Out-File $filepath -Append

    '$passwordSecure = ConvertTo-SecureString -String $password -Key →
    ([Byte[]]$key.Split(" "))' |
    Out-File $filepath -Append
    ('$cred = New-Object system.Management.Automation.PSCredential("{0}", →
    $passwordSecure)' -f $username) |
    Out-File $filepath -Append
    '$cred' | Out-File $filepath -Append

    notepad $filepath
}

Create-EncryptedCredentialScript
```

Wenn Sie die Funktion aufrufen, werden Sie nach einem Benutzernamen und einem Kennwort gefragt. Anschließend generiert die Funktion ein neues Skript und öffnet es im Editor. Darin enthalten ist der Code, um ein verschlüsseltes Kennwort in ein *Credential*-Objekt zu verwandeln. Ein solches Skript sieht danach folgendermaßen aus:

```
$password =
"76492d1116743f0423413b16050a5345MgB8AHcALwBNAHkARABPAEMATgBQAGkANABjAHUAYgBRAGoAWAAzAFAA
YQAxAEeAPQA9AHwAOQAxADEANwAxADEAOAAwAGEANAA3ADAAZAA2AGUAMABhADgANgBjADAAZgBkADgANABhADkAM
AAxAGQAYwBjADEAYQAzADQANwBiAGUAMQBjADgAYwA5ADIANAB1AGUAYgA2AGUA0AAyADcANAA5AGMANwA3AGYAMw
AxADYAYgA="
$key = "247 177 45 190 235 212 54 3 213 180 225 186 93 127 84 90 162 240 155 110 34 74 226
248 182 254 149 175 116 14 145 222"
$passwordSecure = ConvertTo-SecureString -String $password -Key ([Byte[]]$key.Split(" "))
$cred = New-Object system.Management.Automation.PSCredential("schulung\Tobias",
$passwordSecure)
$cred
```

Natürlich ist auch dieser Ansatz nicht absolut sicher, denn das Kennwort kann nach wie vor mit etwas Fachkenntnis aus den verschlüsselten Informationen entschlüsselt werden. Doch ist dies nun erheblich schwieriger als bei einem Kennwort im Klartext.

Prozess als Administrator starten

Sie arbeiten mit aktivierter Benutzerkontensteuerung (User Access Control, UAC) und möchten einen Prozess mit vollen Administratorrechten starten.

Lösung

Starten Sie den Prozess mit dem besonderen Verb *RunAs*, um die vollen Rechte zu aktivieren.

```
PS > Start-Process powershell.exe -Verb RunAs
```

Hintergrund

Ist die Benutzerkontensteuerung des Betriebssystems aktiv, werden bei der Anmeldung alle Sonderrechte aus dem Zugriffstoken des Anwenders entfernt. Auch wenn Sie beispielsweise Administrator sind, können Sie so aus Sicherheitsgründen keine besonders privilegierten Aktionen ausführen.

Um ein Programm mit vollen Berechtigungen auszuführen, muss der eingeschränkte Anwender die Benutzerkontensteuerung beauftragen, sein ursprüngliches Zugriffstoken zu aktivieren und damit den Prozess zu starten. Dies erreichen Sie, indem Sie den Prozess mit dem Verb *RunAs* ausführen. Dieses Verb hat dieselbe Funktion, als würden Sie manuell einen Prozess über sein Kontextmenü und »Als Administrator ausführen« wählen. Das Verb *RunAs* öffnet auf Systemen, die älter sind als Windows Vista, das *Ausführen als*-Dialogfeld, mit dem man ein Programm im Namen eines anderen Benutzers ausführen kann.

Möchten Sie ein Skript mit vollen Administratorrechten ausführen, verwenden Sie als Programm *powershell.exe* und geben Sie in der Eigenschaft *Arguments* die Argumente für *powershell.exe* an, also beispielsweise den Pfadnamen der auszuführenden Skriptdatei.

WICHTIG Stellen Sie sicher, dass das Skript, das mit erhöhten Rechten ausgeführt werden soll, an einem Ort liegt, für den mindestens Leseberechtigungen bestehen.

```
PS > $arguments = '-noexit -noprofile -command →  
"c:\Users\Public\Documents\test.ps1"  
PS > Start-Process powershell.exe -Verb RunAs -ArgumentList $arguments
```

In diesem Beispiel werden einige der PowerShell-Parameter eingesetzt:

Parameter	Beschreibung
<i>-Command</i>	Alles hinter diesem Parameter wird als Kommando interpretiert, das die PowerShell ausführen soll. Hier könnte zum Beispiel der absolute Pfadname zu einer Skriptdatei stehen. Dieser Parameter muss der letzte Parameter sein, weil alle weiteren Parameter als ausführbarer Code gewertet würden.
<i>-ExecutionPolicy</i>	Legt die Ausführungsrichtlinie für diesen Aufruf fest. Möchten Sie mit dem Parameter <i>-File</i> ein Skript starten, sollten Sie außerdem die Ausführungsrichtlinie mit <i>-ExecutionPolicy Bypass</i> umgehen, weil das Skript andernfalls je nach lokal eingerichteter Ausführungsrichtlinie nicht ausgeführt werden kann. Dieser Parameter kann die Ausführungsrichtlinie nicht ändern, wenn diese per Gruppenrichtlinie zentral vorgegeben wurde.
<i>-File</i>	Pfadname zu einer PowerShell-Skriptdatei. Enthält der Pfadname Leerzeichen, muss er in doppelte (nicht in einfachen »'«) Anführungszeichen gestellt werden.
<i>-NoExit</i>	Konsole soll nach Abarbeiten der Befehle geöffnet bleiben. Dadurch bleiben die in die Konsole ausgegebenen Informationen sichtbar. Allerdings empfängt die Konsole nun interaktiv weitere Befehle, die mit vollen Administratorberechtigungen ausgeführt werden.
<i>-NoLogo</i>	Blendet die Copyright-Angabe aus
<i>-NoProfile</i>	Die Profilskripts des Anwenders werden nicht geladen. Das beschleunigt den Start erheblich. Da die meisten Profilskripts dazu dienen, Befehlserweiterungen zu laden, müssen Sie allerdings sicher sein, dass das Skript, das Sie ausführen, diese zusätzlichen Befehle nicht benötigt.
<i>-STA</i>	Aktiviert den STA-Modus, der für Skripts notwendig ist, die grafische Benutzeroberflächen, Windows Presentation Foundation (WPF) oder bestimmte Systemdialoge verwenden
<i>-WindowStyle</i>	Legt die Größe des Konsolenfensters fest. Erlaubte Angaben sind <i>Normal</i> , <i>Minimized</i> , <i>Maximized</i> und <i>Hidden</i> . Weil PowerShell diese Einstellung erst nach seinem Start umsetzt, blinzelt das Fenster bei der Einstellung <i>Hidden</i> dennoch kurz auf.

Tabelle 9.1 PowerShell-Parameter zum Ausführen von Skripts

TIPP

Powershell.exe unterstützt weitere Parameter für andere Zwecke. Eine ausführliche Aufstellung sämtlicher Parameter erhalten Sie mit diesem Befehl:

```
PS > powershell.exe /?
```

Prozess beenden

Sie möchten einen oder mehrere Prozesse sofort abbrechen.

Lösung

Beenden Sie Prozesse mit *Stop-Process*. Sie können dieses Cmdlet entweder direkt einsetzen, indem Sie den Namen des Prozesses angeben, den Sie beenden wollen:

```
PS > Stop-Process -name Notepad -whatIf
```

```
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (5600)".
```

```
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (8344)".
```

Der Parameter *-whatIf* simuliert den Vorgang aus Sicherheitsgründen nur. Prozesse werden erst gestoppt (getötet), wenn Sie diesen Parameter weglassen.

Oder Sie wählen die Prozesse, die Sie stoppen wollen, zuerst mit *Get-Process* aus und leiten das Ergebnis dann über die Pipeline an *Stop-Process* weiter. Die folgende Zeile würde alle Prozesse der Firma »Lenovo« beenden, also zum Beispiel alle zusätzlich installierten Dienstprogramme eines Lenovo-Notebooks:

```
PS > Get-Process | Where-Object { $_.Company -like 'lenovo*' } | →  
Stop-Process -whatIf
```

```
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "cssauth (2676)".
```

```
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "EZEJMNAP (2652)".
```

```
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "tvtpwm_tray (4012)".
```

Hintergrund

Prozesse, die mit *Stop-Process* beendet werden, erhalten keine Gelegenheit mehr, sich selbst geordnet zu beenden. Bei Prozessen mit interaktiver Oberfläche (zum Beispiel eine Textverarbeitung) erhält der Anwender auch keine Warnung, eventuell ungesicherte Daten zu speichern. Ungesicherte Daten gehen sofort verloren. Darüber hinaus sind Anwendungen nicht mehr in der Lage, geladene DLL-Bibliotheken zu entladen, sodass es zu sogenannten *Memory Leaks* (Arbeitsspeicherverlust) kommen kann: Der Speicher des Computers wird nicht vollständig freigegeben. Deshalb sollten Sie *Stop-Process* nur verwenden, wenn es keine andere Möglichkeit gibt, Prozesse zu beenden – beispielsweise, weil Prozesse sich aufgehängt haben und nicht mehr reagieren.

Stop-Process kann Prozesse auf der Basis ihres Namens oder ihrer Prozess-ID beenden. Nur die Prozess-ID ist eindeutig. Geben Sie stattdessen einen Namen wie »notepad« an, sind alle Notepad-Prozesse betroffen. Deshalb sollten Sie *Stop-Process* zuerst mit dem Parameter *-whatIf* testen, um herauszufinden, ob er tatsächlich nur die gewünschten Prozesse beendet. Mit dem Parameter *-confirm* können Sie das Beenden eines Prozesses jeweils einzeln bestätigen oder ablehnen:

```
PS > Stop-Process -name note* -confirm
```

```
Bestätigung
```

```
Möchten Sie diese Aktion wirklich ausführen?
```

```
Ausführen des Vorgangs "Stop-Process" für das Ziel "notepad (5600)".
```

```
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe (Der Standardwert ist "J"):
```

Geben Sie einen Prozessnamen (ohne Platzhalterzeichen) an, der gegenwärtig nicht ausgeführt wird, erhalten Sie einen Fehler, den Sie aber mit dem Parameter *-errorAction* (kurz *-ea*) ausblenden können:

```
PS > Stop-Process -name notepad -ea SilentlyContinue -confirm
```

Abgestürzte Prozesse finden und beenden

Sie möchten alle Prozesse sehen, die nicht mehr reagieren, und diese Prozesse beenden.

Lösung

Lassen Sie sich alle Prozesse ausgeben, bei denen die Eigenschaft *Responding* den Wert *\$false* hat, und leiten Sie die Prozesse an *Stop-Process* weiter:

```
PS > Get-Process | ? { $_.Responding -eq $false } | Stop-Process -confirm
```

Hintergrund

Windows überprüft kontinuierlich, ob Programme noch auf Nachrichten reagieren. Reagieren Programme nicht mehr, sind sie »eingefroren«, reagieren also auch nicht mehr auf Benutzereingaben und aktualisieren ihren Fensterinhalt nicht.

Ein nicht mehr reagierendes Programm muss nicht automatisch bedeuten, dass das Programm abgestürzt und die darin noch ungespeicherten Daten unwiederbringlich verloren sind. Manchmal sind Programme nur ausgelastet und wenn man ihnen ein wenig Zeit gibt, fangen sie sich wieder. Meist deuten nicht mehr reagierende Programme allerdings auf Programmfehler hin.

Die Prozesse, die *Get-Process* liefert, enthalten die Eigenschaft *Responding*. Sie ist *\$false*, wenn Windows das Programm für nicht mehr reagierend erklärt. Damit ist diese Eigenschaft gleichbedeutend mit dem Zusatz »Reagiert nicht mehr«, den man hinter solchen Programmen im Task-Manager sieht.

Abkürzungen für häufige Befehle einrichten

Sie wollen eine Anwendung, die Sie häufiger benötigen, unter einem kürzeren oder leichter zu merkenden Namen aufrufen.

Lösung

Richten Sie mit *Set-Alias* einen Alias auf den gewünschten Befehl ein. Den Internet Explorer starten Sie künftig mit dem neuen Aliasbefehl *ie*:

```
PS > Set-Alias ie "$env:programfiles\Internet Explorer\iexplore.exe"  
PS > ie
```

Hintergrund

Aliasnamen dienen der Bequemlichkeit. PowerShell definiert viele Aliasnamen automatisch, die es Ihnen ermöglichen, mit den gewohnten Konsolenbefehlen weiterzuarbeiten.

Rufen Sie zum Beispiel den Befehl *dir* auf, wird in Wirklichkeit das Cmdlet *Get-ChildItem* ausgeführt. Auch der in der Unix-Welt bekanntere Befehl *ls* ruft in Wirklichkeit *Get-ChildItem* auf. Mit *Set-Alias* legen Sie selbst eigene Aliasnamen an. Allerdings gelten neu hinzugefügte Aliasnamen nur, bis Sie PowerShell beenden.

Wollen Sie neu hinzugefügte Aliasnamen dauerhaft einrichten, fügen Sie die *Set-Alias*-Anweisungen in eines Ihrer Profilskripts ein, die automatisch bei jedem Start der PowerShell ausgeführt werden. Sie können Aliasdefinitionen auch mit *Export-Alias* in eine Datei exportieren und mit *Import-Alias* später (oder auf einem anderen System) wieder einlesen.

Welche Aliasnamen es gibt, ermittelt *Get-Alias*.

PS > **Get-Alias**

CommandType	Name	Definition
-----	----	-----
Alias	ac	Add-Content
Alias	asnp	Add-PSSnapin
Alias	clc	Clear-Content
Alias	cli	Clear-Item
Alias	clp	Clear-ItemProperty
Alias	clv	Clear-Variable
Alias	cpi	Copy-Item
Alias	cpp	Copy-ItemProperty
(...)		

Wenn Sie herausfinden möchten, welche Aliasnamen für ein bestimmtes Cmdlet eingerichtet sind, filtern Sie die Liste anhand der *Definition*-Eigenschaft:

PS > **Get-Alias -Definition 'Get-ChildItem'**

CommandType	Name	Definition
-----	----	-----
Alias	gci	Get-ChildItem
Alias	ls	Get-ChildItem
Alias	dir	Get-ChildItem

Ein Alias kann nur Stellvertreter für einen anderen Befehlsnamen sein, aber keine sonstigen Dinge wie zum Beispiel Standardargumente umfassen. Möchten Sie zusätzlich zum Befehlsnamen weitere Angaben machen, benötigen Sie anstelle eines Alias eine Funktion. Die folgende Funktion ruft *ping.exe* mit einigen Parametern auf, die dafür sorgen, dass anstelle von vier Pings nur ein Ping ausgesendet wird und die Timeoutzeit auf 100 Millisekunden festgelegt ist:

```
PS > Function Ping { ping.exe -n 1 -w 100 $args }  
PS > Ping 10.10.10.10
```

ACHTUNG In der Suchreihenfolge haben Aliasnamen Priorität vor Funktionsnamen. Wenn es also bei Ihnen bereits einen Alias für *Ping* gibt, wird Ihre Funktion nicht mehr aufgerufen. So können Sie einen Alias entfernen:

```
PS > del Alias:Ping
```

Ausgaben eines Programms weiterverarbeiten

Sie möchten die Textausgabe eines konsolenbasierten Programms in PowerShell weiterverarbeiten.

Lösung

Speichern Sie das Ergebnis der konsolenbasierten Anwendung in einer Variablen:

```
PS > $ergebnis = ipconfig -all  
PS > $ergebnis  
  
Windows-IP-Konfiguration  
  
    Hostname . . . . . : LT1  
    Primäres DNS-Suffix . . . . . :  
    Knotentyp . . . . . : Hybrid  
    IP-Routing aktiviert . . . . . : Nein  
(...)  
    Verbindungslokale IPv6-Adresse . . : fe80::1845:3461:3f57:fd96%17(Bevorzugt)  
    Standardgateway . . . . . : ::  
    NetBIOS über TCP/IP . . . . . : Deaktiviert  
  
PS > $ergebnis | Select-String 'NetBIOS'  
  
    NetBIOS über TCP/IP . . . . . : Aktiviert  
    NetBIOS über TCP/IP . . . . . : Aktiviert  
    NetBIOS über TCP/IP . . . . . : Deaktiviert  
  
PS > $ergebnis | ForEach-Object { $_ -like '*NetBIOS*' }  
  
    NetBIOS über TCP/IP . . . . . : Aktiviert  
    NetBIOS über TCP/IP . . . . . : Aktiviert  
    NetBIOS über TCP/IP . . . . . : Deaktiviert
```

Falls die konsolenbasierte Anwendung einen Rückgabewert geliefert hat, können Sie auch diesen Rückgabewert auswerten. Er steht in der Variablen *\$lastexitcode* zur Verfügung:

```
PS > ping.exe 10.10.10.10
```

```
Ping wird ausgeführt für 10.10.10.10 mit 32 Bytes Daten:
Zeitüberschreitung der Anforderung.
(...)
```

```
PS > $lastexitcode
1
```

Hintergrund

Wenn Sie eine konsolenbasierte Anwendung von PowerShell aus starten, empfängt PowerShell automatisch das Ergebnis und verwertet es. Dabei interpretiert PowerShell das Zeilenumbruchzeichen als Beginn einer neuen (Text)-Zeile. Das Ergebnis besteht also aus einem Feld mit mehreren Textzeilen, die sich deshalb mit Cmdlets wie *Select-String* filtern lassen.

```
PS > $ergebnis = ipconfig -all
```

```
PS > $ergebnis.Count
110
```

```
PS > $ergebnis[3]
    Hostname . . . . . : LT1
```

```
PS > $ergebnis | Select-String "NetBIOS"

NetBIOS über TCP/IP . . . . . : Aktiviert
NetBIOS über TCP/IP . . . . . : Aktiviert
NetBIOS über TCP/IP . . . . . : Deaktiviert
```

Select-String liefert allerdings keine Textobjekte zurück, sondern *MatchInfo*-Objekte, weil es intern mithilfe von regulären Ausdrücken die zutreffenden Zeilen ermittelt. Mehr Kontrolle erhalten Sie mit einer einfachen *Where-Object*-Bedingung. Sie liefert die reinen Textzeilen zurück, die Ihrem Kriterium entsprechen.

Möchten Sie das Ergebnis eines nativen Befehls ungefiltert und unbearbeitet empfangen, rufen Sie die native Anwendung selbst auf und überlassen diese Aufgabe nicht PowerShell:

```
Function Run-NativeApp([string] $cmd, [string] $arguments) {
    $processStartInfo = New-Object System.Diagnostics.ProcessStartInfo
    $processStartInfo.FileName = $cmd
    $processStartInfo.WorkingDirectory = (Get-Item (Get-Location)).FullName
    if ($arguments) { $processStartInfo.Arguments = $arguments }
    $processStartInfo.UseShellExecute = $false
    $processStartInfo.RedirectStandardOutput = $true
    $process = [System.Diagnostics.Process]::Start($processStartInfo)
    $process.WaitForExit()
    $process.StandardOutput.ReadToEnd()
}
```

Wenn Sie nun denselben nativen Befehl mithilfe von *Run-NativeApp* ausführen, erhalten Sie diesmal das vollkommen ungefilterte Ergebnis in Form eines einzelnen Strings zurück:

```
PS > $ergebnis = Run-NativeApp ipconfig -all
PS > $ergebnis.GetType().Name
String
```

Verwenden Sie den Umweg über *Run-NativeApp*, wenn die native Anwendung Binärdaten liefert.

Wollen Sie nur wissen, ob die Anwendung fehlerfrei ausgeführt wurde, werten Sie nach dem Aufruf der nativen Anwendung die besondere Variable *\$lastexitcode* aus. Dieser Rückgabewert wird auch »Errorlevel« genannt.

Was der Zahlenwert in dieser Variablen tatsächlich bedeutet, hängt von der Anwendung ab, die Sie zuvor gestartet haben, und wird von ihr bestimmt. In den meisten Fällen bedeutet ein Rückgabewert 0, dass die Anwendung fehlerfrei ausgeführt wurde. Die Konsolenanwendung *ping.exe* beispielsweise meldet einen Wert 0 zurück, wenn das angegebene Ziel erreichbar war, und eine 1, wenn nicht. So könnten Sie eine Funktion schreiben, die prüft, ob bestimmte Systeme online sind oder nicht.

```
Function isOnline([string]$ip) {
    ping.exe $ip -n 1 -w 500 > $null
    ($lastexitcode -eq 0)
}
```

```
PS > isOnline 10.10.10.10
False
PS > isOnline 127.0.0.1
True
PS > isOnline www.tagesschau.de
True
```

ACHTUNG Dieser Ansatz ist begrenzt durch die Limitationen von *ping.exe*. Liegt die IP-Adresse außerhalb Ihres Netzwerksegments und meldet *ping.exe* deshalb »Zielhost nicht erreichbar«, wird dennoch eine erfolgreiche 0 zurückgemeldet, denn in diesem Fall hat *ping.exe* ja eine Antwort erhalten, wenn auch von einem Router. Aus Performancegründen führt *isOnline()* den *Ping*-Befehl mit den Optionen *-n 1* und *-w 500* aus, führt also nur einen Ping durch und wartet maximal 500 Millisekunden. Diese Werte müssten Sie gegebenenfalls an Ihre Netzwerksituation anpassen. Schließlich erkennt *ping.exe* nur Systeme, die auf den Ping antworten, also ICMP-Requests beantworten. Gibt sich ein System auf diese Weise nicht zu erkennen, kann es trotzdem online sein.

So könnten Sie sehr einfach einen Segmentscan durchführen:

```
Function isOnline([string]$ip) {
    ping.exe $ip -n 1 -w 100 > $null
    ($lastexitcode -eq 0)
}

PS > foreach ($x in 1..2) {
>> foreach ($y in 0..255) {
>> $ip = "192.168.$x.$y"
>> $online = isOnline($ip)
>> "$ip = $online"
>> }
>> }
>>
192.168.1.0 = False
192.168.1.1 = True
192.168.1.2 = True
(...)
```

Alternativ kann auch das Cmdlet *Test-Connection* eingesetzt werden.


Zusammenfassung

Laufende Programme werden *Process* genannt und so stehen für die Verwaltung von Programmen die Cmdlets der Familie *Process* zur Verfügung:

```
PS > Get-Command -noun Process
```

CommandType	Name	Definition
Cmdlet	Debug-Process	Debug-Process [-Name] <String[]> [-Verbose]...
Cmdlet	Get-Process	Get-Process [[-Name] <String[]>] [-Computer...
Cmdlet	Start-Process	Start-Process [-FilePath] <String> [[-Argum...
Cmdlet	Stop-Process	Stop-Process [-Id] <Int32[]> [-PassThru] [-...
Cmdlet	Wait-Process	Wait-Process [-Name] <String[]> [[-Timeout]...

Die drei wichtigsten Cmdlets sind *Get-Process* (laufende Prozesse untersuchen), *Start-Process* (neuen Prozess starten) und *Stop-Process* (laufenden Prozess abbrechen).

Zwar brauchen Sie zum Starten eines Prozesses in PowerShell eigentlich gar kein separates Cmdlet wie *Start-Process*. Es genügt, den Pfad zur ausführbaren Datei anzugeben und die -Taste zu drücken. In diesem Fall wird der Prozess allerdings stets nach festen Regeln gestartet:

- Der Prozess läuft unter dem Benutzerkonto des Aufrufers
- Konsolenbasierte Programme werden synchron ausgeführt und teilen sich das Konsolenfenster mit PowerShell

- Fensterbasierte Programme werden asynchron ausgeführt. PowerShell wartet nicht auf das Ende solcher Prozesse, sondern fährt sofort fort.

Möchten Sie von diesen Regeln abweichen oder zusätzliche Feinheiten wie die Fenstergröße bestimmen, bietet *Start-Process* hierfür mit seinen Parametern die Möglichkeit.

Get-Process liefert die Prozessobjekte der laufenden Prozesse zurück. Geben Sie diese in der Konsole aus, werden nur die wichtigsten Informationen angezeigt. Sämtliche Informationen erhalten Sie, indem Sie das Ergebnis an *Select-Object* * weiterleiten:

```
PS > Get-Process powershell | Select-Object *
```

Die Prozessobjekte, die Sie von *Get-Process* erhalten, verfügen außerdem über Methoden (Befehle), mit denen Sie die Prozesse ebenfalls steuern können. Diese Methoden sind nur dann nicht (mehr) vorhanden, wenn Sie die Prozessobjekte bereits umgewandelt haben, also beispielsweise nach Einsatz von *Select-Object*, oder falls Sie die Prozesse via PowerShell-Remoting von einem Remotesystem abgerufen haben.

```
PS > notepad
PS > Get-Process notepad | Get-Member -memberType *method
```

```
TypeName: System.Diagnostics.Process
```

Name	MemberType	Definition
(...)		
Close	Method	System.Void Close()
CloseMainWindow	Method	bool CloseMainWindow()
(...)		
Kill	Method	System.Void Kill()
(...)		
WaitForExit	Method	bool WaitForExit(int millisecond...)
WaitForInputIdle	Method	bool WaitForInputIdle(int millis...)

Um beispielsweise sämtliche Instanzen von Notepad zu schließen, aber dem Anwender dabei die Möglichkeit zu lassen, ungespeicherte Arbeiten vorher abzuspeichern, rufen Sie *CloseMainWindow()* auf:

```
PS > Get-Process notepad | ForEach-Object { $_.CloseMainWindow() }
```

Möchten Sie einen bestimmten Prozess ansprechen, benötigen Sie dessen eindeutige Prozess-ID:

```
PS > Get-Process -Id $pid
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
901	130	583788	74572	1025	145,72	952	PowerShell

Oder Sie beauftragen *Start-Process* direkt beim Start mit dem Parameter *-passThru*, das zugrunde liegende *Process*-Objekt des gestarteten Prozesses zurückzuliefern. Dieses könnten Sie dann in einer Variablen speichern, um den Prozess später zu steuern. Die folgende Zeile öffnet den Windows-Editor und speichert das Prozessobjekt:

```
PS > $notepad = Start-Process notepad -passThru
```

Später könnten Sie diesen Prozess gezielt jederzeit sofort schließen oder dem Anwender die Möglichkeit zum Speichern geben:

```
PS > $notepad.Close()  
PS > $notepad.CloseMainWindow()
```

Kapitel 10

Dienste

In diesem Kapitel:

Alle Dienste auflisten	316
Bestimmte Dienste finden	318
Dienste starten oder stoppen	321
Abhängige Dienste finden	323
Dienste-Einstellungen ändern	324
Auf eine Dienststatusänderung warten	326
Uptime eines Diensts bestimmen	327
Zugrunde liegende Dienstprogramme ermitteln	329
Dienste-Informationen als CSV in Excel importieren	330
Dienstzustand überprüfen	331
Neuen Dienst installieren	332
Dienst entfernen	332
Zusammenfassung	333

Dienste sind besondere Programme, die nicht an einen bestimmten Benutzer gebunden sind, sondern automatisch oder bei Bedarf vom System gestartet und ausgeführt werden. PowerShell kann Dienste überwachen und steuern.

Korrekt konfigurierte Dienste sind für den reibungslosen Betrieb des Computers essentiell. Deshalb erfordern die meisten Änderungen an den Dienste-Einstellungen Administratorrechte. Der jeweilige Dienststatus kann dagegen auch ohne besondere Rechte gelesen werden.

Nehmen Sie keine unbedachten Änderungen an Diensten vor, da diese weitreichende Folgen haben können.

Alle Dienste auflisten

Sie möchten alle Dienste auflisten, die auf Ihrem System installiert sind.

Lösung

Setzen Sie *Get-Service* ein:

```
PS > Get-Service
```

Status	Name	DisplayName
-----	----	-----
Running	AEADIFilters	Andrea ADI Filters Service
Running	AeLookupSvc	Anwendungserfahrung
Stopped	ALG	Gatewaydienst auf Anwendungsebene
(...)		

Sie können auch einen Teil des Dienstnamens angeben, um die Ausgabe auf bestimmte Dienste zu beschränken:

```
PS > Get-Service W32*
```

Status	Name	DisplayName
-----	----	-----
Running	W32Time	Windows-Zeitgeber

Wollen Sie alle Informationen sehen, die PowerShell über Dienste liefert, senden Sie das Ergebnis an *Select-Object* *:

```
PS > Get-Service W32* | Select-Object *
```

```
Name           : W32Time
RequiredServices : {}
CanPauseAndContinue : False
CanShutdown     : True
CanStop         : True
DisplayName      : Windows-Zeitgeber
```

```

DependentServices : {}
MachineName      : .
ServiceName      : W32Time
ServicesDependedOn : {}
ServiceHandle    :
Status           : Running
ServiceType      : Win32ShareProcess
Site             :
Container        :

```

Noch ausführlichere Informationen zu Diensten liefert die WMI (Windows Management Instrumentation):

```
PS > Get-WmiObject Win32_Service -filter 'name like "W32%"'
```

```

ExitCode : 0
Name     : W32Time
ProcessId : 0
StartMode : Manual
State    : Stopped
Status   : OK

```

```
PS > Get-WmiObject Win32_Service -filter 'name like "W32%"' | Select-Object *
```

```

Name           : W32Time
Status         : OK
ExitCode       : 0
DesktopInteract : False
ErrorControl   : Normal
PathName       : C:\Windows\system32\svchost.exe -k
                LocalService
ServiceType    : Share Process
StartMode      : Manual
(...)
Description    : Behält Datums- und Zeitsynchronisation auf
                allen Clients und Servern im Netzwerk
                bei. Wird dieser Dienst beendet, sind Datums-
                und Zeitsynchronisation nicht verfügbar.
                Wird dieser Dienst deaktiviert, können keine
                explizit von der Synchronisation abhängigen
                Dienste gestartet werden.
DisplayName    : Windows-Zeitgeber
InstallDate    :
ProcessId      : 0
ServiceSpecificExitCode : 0
Started       : False
StartName      : NT AUTHORITY\LocalService
State         : Stopped
(...)

```

Hintergrund

Get-Service liefert Dienste als *ServiceController*-Objekte zurück. Diese Objekte enthalten die wichtigsten Informationen zu den Diensten und informieren im Wesentlichen, ob ein Dienst gerade ausgeführt wird oder nicht. Aussagekräftiger ist das *Win32_Service*-Objekt, das Sie mit *Get-WmiObject* vom WMI-Dienst abrufen. Es liefert nicht nur wesentlich mehr Informationen, es kann auch remote Dienste anderer Computer über das Netzwerk abfragen und bietet dabei eine Anmelde­möglichkeit:

```
PS > Get-WmiObject Win32_Service -computername PC00223 -Credential Administrator
```

Get-Service ermöglicht zwar ebenfalls mit dem Parameter *-ComputerName* den Remotezugriff, doch ist eine Anmeldung unter alternativem Benutzerkonto hier nicht möglich. Zudem funktioniert dieser Remotezugriff nur, wenn auf dem Zielsystem der Dienst *RemoteRegistry* ausgeführt wird. Falls nicht, kommt es zu Fehlermeldungen.

Bestimmte Dienste finden

Sie möchten bestimmte Dienste finden, zum Beispiel alle ausgeführten Dienste oder Dienste mit einem besonderen Startverhalten.

Lösung

Suchen Sie einen bestimmten Dienst, verwenden Sie Platzhalterzeichen. Möchten Sie nach dem deutschsprachigen Dienstnamen suchen, greifen Sie zum Parameter *-DisplayName*:

```
PS > Get-Service *spooler*

Status  Name          DisplayName
-----  ----          -
Stopped Spooler       Druckwarteschlange

PS > Get-Service *druck*
PS > Get-Service -DisplayName *druck*

Status  Name          DisplayName
-----  ----          -
Stopped Spooler       Druckwarteschlange
```

Möchten Sie alle Dienste sehen, die gerade ausgeführt werden, filtern Sie mit *Where-Object* nach der Eigenschaft *Status*:

```
PS > Get-Service | Where-Object { $_.Status -eq 'Running' }
```

Geben Sie bei WMI-Abfragen mit *Get-WmiObject* Ihr Kriterium als serverseitigen WMI-Filter an, was insbesondere bei Remoteabfragen effizienter ist als die clientseitige Filterung von *Where-Object*. So finden Sie alle laufenden Dienste mit Hilfe der WMI:

```
PS > Get-WmiObject Win32_Service -filter 'Started=True'
```

Möchten Sie alle Autostart-Dienste finden, die aber nicht ausgeführt werden, und deren Exit-Code analysieren, wäre diese Abfrage hilfreich:

```
PS > Get-WmiObject Win32_Service -filter 'Started=False and Startmode="Auto"' | →
    Select-Object Name, Caption, ExitCode
```

Name	Caption	ExitCode
----	-----	-----
clr_optimization_v4.0.30319_32	Microsoft .NET Framework NGEN ...	0
clr_optimization_v4.0.30319_64	Microsoft .NET Framework NGEN...	0
gupdate	Google Update Service (...)	0
MMCSS	Multimediaklassenplaner	0
Pml Driver HPZ12	Pml Driver HPZ12	0
spsvc	Software Protection	0

Ein ExitCode ungleich 0 würde auf einen Problemfall hinweisen. Alle gestoppten Dienste mit einem ExitCode ungleich 0 finden Sie so:

```
PS > Get-WmiObject Win32_Service -filter 'Started=False and ExitCode<>0' | →
    Select-Object Name, Caption, ExitCode
```

TIPP

Ein sehr häufig vorkommender harmloser Rückgabecode lautet *1077*: Seit dem letzten Reboot wurde nicht versucht, diesen Dienst zu starten. Um tatsächliche Problemfälle zu finden, sollten Sie diesen Exit-Code also ausschließen:

```
PS > Get-WmiObject Win32_Service -filter 'Started=False and ExitCode<>0 and →
    ExitCode<>1077' | Select-Object Name, Caption, ExitCode
```

Hintergrund

Entscheiden Sie sich zuerst für einen der zwei Zugangswege zu Diensten: Verwenden Sie entweder *Get-Service* oder *Get-WmiObject Win32_Service*. Beispielsweise könnten Sie mit *Get-Service* leicht eine Tabelle mit der Zuordnung zwischen deutschem und internationalem Dienstenamen herstellen:

```
PS > Get-Service | Sort-Object Name | ForEach-Object { "{0,-20} = {1,-30}" -f →
    $_.Name, $_.DisplayName }
```

AeLookupSvc	= Anwendungserfahrung
ALG	= Gatewaydienst auf Anwendungsebene
AppIDSvc	= Anwendungsidentität
Appinfo	= Anwendungsinformationen
Apple Mobile Device	= Apple Mobile Device
AppMgmt	= Anwendungsverwaltung
arXfrSvc	= Windows Media Center TV Archive Transfer Service
aspnet state	= ASP.NET-Zustandsdienst
AudioEndpointBuilder	= Windows-Audio-Endpunkterstellung
AudioSrv	= Windows-Audio
AxInstSV	= ActiveX-Installer (AxInstSV)
(...)	

Sobald Sie mehr über einen Dienst wissen wollen, beispielsweise seinen Startmodus, greifen Sie auf die WMI zu. Die folgende Zeile liefert alle Dienste, die automatisch starten, aber nicht (mehr) ausgeführt werden:

```
PS > Get-WmiObject Win32_Service -filter 'StartMode="Auto" and Started=False' | →
      Select-Object Caption, StartMode, State, ExitCode
```

Caption -----	StartMode -----	State -----	ExitCode -----
Microsoft .NET F...	Auto	Stopped	0
Microsoft .NET F...	Auto	Stopped	0
Google Update Se...	Auto	Stopped	0
Multimediaklasse...	Auto	Stopped	0
Pml Driver HPZ12	Auto	Stopped	0
Druckwarteschlange	Auto	Stopped	0
Software Protection	Auto	Stopped	0

TIPP

Bei Problemfällen besonders interessant ist der Rückgabewert eines Dienstes, denn wenn dieser ungleich 0 ist, verrät er die Problemursache. Allerdings ist der Rückgabewert eine codierte Zahl. Um die Klartextmeldung des Fehlers zu sehen, können Sie diese Zahl aber an *net.exe* weiterleiten:

```
PS > net helpmsg 1077
```

Seit dem letzten Systemstart wurde nicht versucht, den Dienst zu starten.

```
PS > $klartext = @{
>> Name = 'Fehlermeldung'
>> Expression = { (net.exe helpmsg $_.ExitCode)[1] }
>> }
>>
```

```
PS > Get-WmiObject Win32_Service -filter 'ExitCode<>0' | →
      Select-Object Caption, $klartext | Format-Table -AutoSize -Wrap
```

Caption -----	Fehlermeldung -----
Anwendungsidentität	Seit dem letzten Systemstart wurde nicht versucht, den Dienst zu starten.
Anwendungsverwaltung	Seit dem letzten Systemstart wurde nicht versucht, den Dienst zu starten.
ASP.NET-Zustandsdienst	Seit dem letzten Systemstart wurde nicht versucht, den Dienst zu starten.
(...)	

Get-Service liefert nur Anwendungsdienste. Gerätetreiberdienste liefert das Cmdlet nicht. Diese können Sie aber über die .NET-Methode *GetDevices()* ebenfalls sichtbar machen:


```
PS > [System.ServiceProcess.ServiceController]::GetDevices()
```

Status	Name	DisplayName
-----	----	-----
Running	ACPI	Microsoft ACPI-Treiber
Running	ADIHdAudAddService	ADI UAA Function Driver for High De...
Stopped	adp94xx	adp94xx
Stopped	adpahci	adpahci
Stopped	adpu160m	adpu160m
Stopped	adpu320	adpu320
Running	AFD	Ancilliary Function Driver for Winsock
Stopped	agp440	Intel AGP Bus Filter
(...)		

Dienste starten oder stoppen

Sie möchten einen Dienst starten, stoppen, anhalten oder neu starten.

Lösung

Um einen Dienst zu starten, verwenden Sie *Start-Service* und geben den gewünschten Dienst entweder mit seinem englischen Universalnamen oder mit seinem landestypischen Anzeigenamen an. Im Beispiel wird der Dienst *Spooler* (Drucker-Spooler) gestartet, der allerdings nicht bei allen Windows-Versionen installiert ist:

```
PS > Start-Service spooler
```

HINWEIS

Zum Starten und Stoppen der meisten Dienste benötigen Sie Administratorrechte. Sind andere Dienste von einem Dienst abhängig, kann dieser Dienst nur gestoppt oder neu gestartet werden, wenn Sie den Parameter *-Force* des Cmdlets *Stop-Service* einsetzen:

```
PS > Restart-Service spooler -force
```

Das Anhalten oder Beenden eines Diensts kann die Stabilität des Computers oder Ihrer Programme beeinträchtigen. Beenden Sie also nicht wahllos beliebige Dienste, nur um zu sehen, was als Nächstes passiert.

Hintergrund

PowerShell bietet eine Reihe von Service-Cmdlets für die Dienstverwaltung, die man sich mit *Get-Command* auflisten lassen kann. Verwenden Sie als Parameter *-noun* das gemeinsame Substantiv aller Cmdlets rund um Dienste, nämlich *Service*:

```
PS > Get-Command -noun Service
```

CommandType	Name	Definition
Cmdlet	Get-Service	Get-Service [[-Name] <String[]>] [-Includ...
Cmdlet	New-Service	New-Service [-Name] <String> [-BinaryPath...
(...)		

Die folgenden Cmdlets stehen für die Verwaltung von Diensten zur Verfügung:

Name	Beschreibung
<i>Get-Service</i>	Listet einen oder mehrere Dienste auf
<i>New-Service</i>	Installiert einen neuen Dienst
<i>Restart-Service</i>	Startet einen Dienst neu, indem der Dienst zuerst gestoppt und danach erneut gestartet wird
<i>Resume-Service</i>	Setzt einen zuvor angehaltenen Dienst fort
<i>Set-Service</i>	Ändert den Namen, die Beschreibung oder den Startmodus eines Diensts
<i>Start-Service</i>	Startet einen Dienst
<i>Stop-Service</i>	Beendet einen Dienst
<i>Suspend-Service</i>	Hält einen Dienst vorübergehend an

Tabelle 10.1 Cmdlets zur Verwaltung von Diensten

Zwar könnten Sie Dienste auch über die WMI verwalten, doch stehen dafür keine Cmdlets zur Verfügung. Sie müssten deshalb die entsprechenden Methoden des WMI-Objekts *Win32_Service* direkt aufrufen:

```
PS > (Get-WmiObject Win32_Service -filter →
'<name="wscsvc">').StartService().ReturnValue
0
PS > (Get-WmiObject Win32_Service -filter →
'<caption="Windows Search">').StopService().ReturnValue
0
```

Die wichtigsten Rückgabewerte sind 0, 2 und 10:

- **Erfolgreich** Der Rückgabewert 0 zeigt an, dass die Aktion erfolgreich ausgeführt wurde
- **Dienst läuft bereits** Der Rückgabewert 10 meldet, dass der Dienst bereits ausgeführt wird und deshalb nicht gestartet werden kann
- **Berechtigungen erforderlich** Der Rückgabewert 2 steht für mangelnde Berechtigungen. Sie müssen Administrator sein, um Dienste starten und stoppen zu können.

Abhängige Dienste finden

Sie möchten wissen, welche Dienste von einem bestimmten Dienst abhängig sind (und also beeinträchtigt würden, wenn Sie den Dienst stoppen).

Lösung

Greifen Sie mit *Get-Service* auf den Dienst zu und rufen Sie seine Eigenschaft *DependentServices* ab. Die folgende Zeile listet alle Dienste auf, die vom *Drucker-Spooler*-Dienst abhängig sind. Diese Dienste müssen nicht gestartet sein, können aber nur gestartet werden, wenn der zugrunde liegende Dienst läuft:

```
PS > $dienst = Get-Service Spooler
PS > $dienst.DependentServices
```

Status	Name	DisplayName
-----	----	-----
Stopped	Fax	Fax

Hintergrund

Ein Dienst kann die Grundlagen für einen anderen Dienst schaffen. Dies nennt man »Abhängigkeit«. Stoppen Sie einen Dienst, sind davon alle abhängigen Dienste betroffen. Sie sollten deshalb einen Dienst erst stoppen, wenn auch seine Abhängigkeiten gestoppt wurden, beispielsweise so:

```
PS > (Get-Service spooler).DependentServices | Stop-Service
```

Umgekehrt kann ein Dienst selbst von anderen Diensten abhängig sein. Welche das sind, erfahren Sie in der Eigenschaft *ServicesDependedOn*:

```
PS > $dienst = Get-Service Spooler
PS > $dienst.ServicesDependedOn
```

Status	Name	DisplayName
-----	----	-----
Running	HTTP	http
Running	RPCSS	Remoteprozeduraufruf (RPC)

Damit also ein Dienst ordnungsgemäß funktionieren kann, müssen alle Dienste gestartet sein, von denen er abhängig ist, beispielsweise so:

```
PS > (Get-Service Spooler).ServicesDependedOn | Start-Service
```

Dienste-Einstellungen ändern

Sie möchten die Grundeinstellungen eines Diensts ändern, zum Beispiel seinen Startmodus oder seine Beschreibung.

Lösung

Verwenden Sie *Set-Service* oder greifen Sie auf die WMI zu. In beiden Fällen benötigen Sie für die meisten Dienste Administratorrechte und sollten sich bewusst sein, welche Auswirkungen Ihre Änderungen haben.

Mit *Set-Service* ändern Sie Anzeigenamen, Beschreibung und Startmodus eines Diensts:

```
PS > Set-Service Fax -DisplayName 'Faxunterstützung' -description →  
      'nötig für Senden und Empfangen von Faxen'  
PS > Set-Service Fax -startupType Automatic
```

Über die WMI führen Sie die Änderung so durch:

```
PS > (Get-WmiObject Win32_Service -filter 'name="Fax"') →  
      .ChangeStartMode('Automatic').ReturnValue  
0  
PS > (Get-WmiObject Win32_Service -filter 'name="Fax"') →  
      .Change('Faxunterstützung').ReturnValue  
0
```

Die Beschreibung des Diensts kann über die WMI nicht geändert werden.

Hintergrund

Das *Win32_Service*-Objekt der WMI liefert zwar 25 Eigenschaften mit interessanten Informationen, jedoch sind diese alle nur lesbar. Änderungen erfolgen bei der WMI fast immer über Methodenaufrufe. Mit den Methoden *ChangeStartMode()* und *Change()* lassen sich alle wesentlichen Diensteeinstellungen ändern. Jedenfalls dann, wenn Sie über die notwendigen Berechtigungen verfügen.

Während Sie über *Set-Service* nur den Anzeigenamen, die Beschreibung und den Startmode ändern können, liefert die WMI auch Informationen über das Benutzerkonto, unter dem ein Dienst ausgeführt wird:

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'name="Fax"'  
PS > $dienst.StartName  
NT AUTHORITY\NetworkService
```

Anmeldekonto und zugehöriges Kennwort ändern Sie über *Change()*:

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'name="SampleService"'
PS > $anmeldung = Get-Credential
PS > $user = $anmeldung.Username
PS > $kennwort = $anmeldung.GetNetworkCredential().Password
PS > $dienst.Change($null,$null,$null,$null,$null,$null,$user,$kennwort) →
    .ReturnValue
0
```

ACHTUNG Das Benutzerkonto, das Sie hier angeben, muss die notwendigen Berechtigungen besitzen, um Dienste zu starten. Andernfalls erhalten Sie den Fehlercode »21: Ungültiger Parameter«. Achten Sie außerdem darauf, dass der Benutzer, in dessen Namen Sie den Dienst starten, auch tatsächlich Zugriffsberechtigungen auf die ausführbare Datei des Diensts besitzt. Andernfalls erhalten Sie beim Start des Diensts einen »Zugriff verweigert«-Fehler.

Change() erwartet für jeden Parameter, den Sie nicht setzen möchten, den Wert *\$null*. Wollen Sie also nur das Kennwort eines Diensts ändern, aber nicht das Benutzerkonto, gehen Sie so vor:

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'name="Fax"'
PS > $kennwort = "StrengGeheim"
PS > $dienst.Change($null,$null,$null,$null,$null,$null,$null,$kennwort) →
    .ReturnValue
0
```

Möchten Sie das Kennwort sämtlicher Dienste ändern, die einem bestimmten Konto zugeordnet sind, gehen Sie folgendermaßen vor:

```
PS > $user = ".\Testuser"
PS > $kennwort = "neuesKennwortGeheim00023"

PS > Get-WmiObject Win32_Service -filter "StartName='$user'" |
>> ForEach-Object {
>> "Kennwort des Dienstes '$($_.Name)' wird geändert:"
>> $code = $_.Change($null,$null,$null,$null,$null,$null,$null,$kennwort) →
>> .ReturnValue
>> if ($code -eq 0) {
>> Write-Host -foregroundColor Green "erfolgreich."
>> } else {
>> Write-Host -foregroundColor Red "nicht erfolgreich, Fehlercode $code"
>> }
>> }
>>
```

Methoden von WMI-Objekten können übrigens nicht nur in der üblichen Methodenschreibweise ausgeführt werden. Alternativ beschaffen Sie sich ein sogenanntes *Parameter*-Objekt, hinterlegen darin die Werte, die der Methode übergeben werden sollen, und rufen die Methode dann über *InvokeMethod()* auf. Dieser Ansatz funktioniert allerdings nicht mit dem von PowerShell zurückgelieferten Dienst-Objekt, sondern nur mit seinem zugrunde liegenden Basisobjekt, das Sie über *PSBase* ansprechen:

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'name="SampleService"'
PS > $inparams = $dienst.PSBase.GetMethodParameters('Change')
PS > $inparams['DisplayName'] = 'Mein neuer Dienst'
PS > $inparams['StartPassword'] = 'StrengGeheim99'
PS > $dienst.PSBase.InvokeMethod('Change', $inparams, $null)
```

Sie können sich diesen Ansatz zunutze machen, um die für eine WMI-Methode erforderlichen Parameter anzuzeigen:

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'name="Fax"'
PS > $dienst.PSBase.GetMethodParameters('ChangeStartMode')
```

Auf eine Dienststatusänderung warten

Sie möchten warten, bis ein Dienst seinen Status in einen bestimmten Zustand ändert.

Lösung

Nutzen Sie die *WaitForStatus()*-Methode des *Dienste*-Objekts. Sie hält das Skript so lange an, bis der Dienst den gewünschten Zustand erreicht oder der angegebene Timeout abläuft.

```
PS > $dienst = Get-Service Spooler
PS > $dienst.Start()
PS > Trap {'Dienst nicht rechtzeitig gestartet!'; break} →
    $dienst.WaitForStatus('Running', (New-Timespan -seconds 10) )
```

Hintergrund

Wenn Sie zum Beispiel einen Dienst starten, kann dieser einige Sekunden benötigen, bis er tatsächlich läuft. Damit Sie den Dienst nicht ansprechen, bevor er funktionstüchtig ist, warten Sie, bis sich sein Zustand tatsächlich geändert hat. Auf diese Weise könnten Sie zum Beispiel einen Dienst neu starten. Diese Funktionalität bietet zwar bereits *Restart-Service*, ist aber ein interessantes Anschauungsobjekt:

```
Function Restart-ServiceEx([string]$dienst = →
    $(Throw( 'Dienstnamen angeben!')) {

    $dienste = @( Get-Service -displayName $dienst | ? →
        { $_.Status -eq 'Running' } )

    switch ($dienste.count) {
        0 { Write-Host -foregroundColor Red 'Kein Dienst gefunden.' }
        1 { Write-Host -foregroundColor Green →
            'Dienst gefunden, wird neu gestartet.' }
        default { Write-Host -foregroundColor yellow →
            'Mehr als ein Dienst gefunden.' }
    }
}
```

```
$dienste | % {  
    Stop-Service $_.Name  
    $_.WaitForStatus('Stopped', (New-TimeSpan -seconds 15))  
    Start-Service $_.Name  
}
```

```
PS > Restart-ServiceEx Spooler
```

Uptime eines Diensts bestimmen

Sie wollen herausfinden, wie lange ein bestimmter Dienst bereits ausgeführt wird.

Lösung

Hinter jedem Dienst steckt ein Programm. Bestimmen Sie, wann dieses Programm gestartet wurde, und berechnen Sie dann die Zeitdifferenz zur aktuellen Uhrzeit:

```
PS > $dienstname = "Spooler"  
PS > $dienst = Get-WmiObject Win32_Service -filter "name='$dienstname'"  
PS > $prozess = Get-WmiObject Win32_Process -filter "→  
    "processid=$( $dienst.processid )"   
PS > $creationtime = ([wmi]'').ConvertToDateTime($prozess.creationDate)  
PS > "Dienst '{0}' wurde gestartet am {1}" -f $dienstname, $creationtime  
Dienst 'Spooler' wurde gestartet am 13.04.2011 15:17:05  
PS > "Laufzeit {0:0} Minuten." -f ((Get-Date) - $creationtime).TotalMinutes  
Laufzeit 1154 Minuten.
```

Hintergrund

Jedes *Win32_Service*-Objekt der WMI liefert in der Eigenschaft *ProcessID* die eindeutige Prozess-ID des Prozesses, der hinter dem Dienst steckt. Ausgerüstet mit dieser *ProcessID* können Sie WMI beauftragen, das zugehörige *Win32_Process*-Objekt zu liefern. Darin findet sich in der Eigenschaft *CreationTime* das Datum und die Uhrzeit, wann dieser Prozess gestartet wurde.

Allerdings liegt diese Zeitangabe im WMI-eigenen Format vor und muss zuerst in eine reguläre Zeitangabe umgerechnet werden. Diese Aufgabe erledigt *ConvertToDateTime*, die von der WMI-Klasse *ManagementObject* bereitgestellt wird. Eine Abkürzung zu dieser Klasse (ein sogenannter *Type Accelerator*) lautet *[wmi]*, und die beiden Anführungszeichen dahinter legen fest, dass Sie das lokale System ansprechen wollen.

Es spielt also keine Rolle, ob Sie den *Type Accelerator* oder den vollen Klassennamen verwenden, um an die Datumsumwandlungsfunktionen heranzukommen. Die folgenden beiden Zeilen konvertieren das aktuelle Datum ins WMI-Format:

```
PS > ([wmi]'').ConvertFromDateTime( (Get-Date) )
20080414103152.311054+120
PS > (new-object System.Management.ManagementObject '' ) →
    .ConvertFromDateTime( (Get-Date) )
20080414103152.353054+120
```

Die Zeitdifferenz, also die Uptime, berechnen Sie schließlich mit dem Subtraktionsoperator »–«, indem Sie die Startzeit des Prozesses von der aktuellen Zeit abziehen. Das Ergebnis ist ein *TimeSpan*-Objekt, das Ihnen die Differenz in verschiedenen Einheiten anbietet. Sie könnten die Uptime also ebenso gut in Sekunden oder Stunden ausgeben:

```
PS > "Laufzeit {0:0} Sekunden." -f ((Get-Date) - $creationtime).TotalSeconds
Laufzeit 69311 Sekunden.
PS > "Laufzeit {0:0} Stunden." -f ((Get-Date) - $creationtime).TotalHours
Laufzeit 19 Stunden.
```

Falls Sie übrigens nicht darauf angewiesen sind, remote auf andere Systeme zuzugreifen, können Sie die Lösung noch vereinfachen und den zugrunde liegenden Dienst mit *Get-Process* beschaffen. So entfällt die Umrechnung der WMI-Zeitangabe:

```
PS > $dienstname = "Spooler"
PS > $dienst = Get-WmiObject Win32_Service -filter "name='$dienstname'"
PS > $prozess = Get-Process -id $dienst.processID
PS > "Dienst '{0}' wurde gestartet am {1}" -f $dienstname, $prozess.StartTime
Dienst 'Spooler' wurde gestartet am 13.04.2011 15:17:05
PS > "Laufzeit {0:0} Minuten." -f ((Get-Date) - $prozess.StartTime).TotalMinutes
Laufzeit 1156 Minuten.
```

Oder noch kürzer (aber unübersichtlicher):

```
PS > $dienstname = "Spooler"
PS > $prozess = Get-Process -id (Get-WmiObject win32_service -filter →
    "name='$dienstname'").processID
PS > "Dienst '{0}' wurde gestartet am {1}" -f $dienstname, $prozess.StartTime
Dienst 'Spooler' wurde gestartet am 13.04.2011 15:17:05
PS > "Laufzeit {0:0} Minuten." -f ((Get-Date) - $prozess.StartTime).TotalMinutes
Laufzeit 1156 Minuten.
```

Der Kern, die Laufzeit eines Diensts in Minuten, kann sogar in einer Zeile zusammengefasst werden:

```
PS > ((Get-Date)-(gps -id (gwmi Win32_Service -f →
    "name='Spooler'").processID).StartTime).TotalMinutes
1156,50825333333
```


Zugrunde liegende Dienstprogramme ermitteln

Sie möchten wissen, welche Programme hinter einem Dienst stecken.

Lösung

Ermitteln Sie die Prozess-ID des Diensts, lassen Sie sich dann den Prozess mit dieser Prozess-ID geben und erfragen Sie dessen *CommandLine*-Eigenschaft. In ihr steht, wie dieser Prozess gestartet wurde:

```
PS > Get-WmiObject Win32_Service | ForEach-Object { $prozess = Get-WmiObject →
    Win32_Process -filter "ProcessID=$(($_.ProcessID)";
>> $prozess | Add-Member NoteProperty ServiceName $_.Name;
>> $prozess} | Format-Table ServiceName, CommandLine
>>
```

ServiceName	CommandLine
-----	-----
AEADIFilters	C:\Windows\system32\AEADISRV.EXE
AeLookupSvc	C:\Windows\system32\svchost.exe -k netsvcs
ALG	
Appinfo	C:\Windows\system32\svchost.exe -k netsvcs
AppMgmt	
AudioEndpointBuilder	C:\Windows\System32\svchost.exe -k LocalSyst...
AudioSrv	C:\Windows\System32\svchost.exe -k LocalServ...
BFE	C:\Windows\system32\svchost.exe -k LocalServ...
(...)	

Hintergrund

Der Code in diesem Beispiel ermittelt zunächst sämtliche Dienste. Ebenso gut könnten Sie sich an dieser Stelle auf die Dienste konzentrieren, die Sie interessieren:

```
PS > Get-WmiObject Win32_Service -filter 'Caption like "Sicherheit%"'
```

```
ExitCode : 0
Name      : SamSs
ProcessId : 752
StartMode : Auto
State     : Running
Status    : OK
```

```
ExitCode : 0
Name      : wscsvc
ProcessId : 1112
StartMode : Auto
State     : Running
Status    : OK
```

Die gefundenen Dienste werden über die Pipeline an eine *ForEach*-Schleife (Kurzform »%«) weitergereicht. Sie ermittelt das *Win32_Process*-Objekt, das dieselbe ProcessID trägt wie im Dienst angegeben und speichert es in *\$prozess*. Im nächsten Schritt legt es *\$prozess* zurück in die Pipeline und sendet es an *Add-Member*. Hier wird dem Prozessobjekt eine neue zusätzliche Eigenschaft namens *ServiceName* hinzugefügt, in der der Name des Diensts vermerkt wird, der in die Pipeline hineingesendet wurde. Auf diese Weise geht die Zuordnung zwischen Dienst und Prozess nicht verloren. Schließlich legt die Schleife *\$prozess* ein letztes Mal zurück auf die Pipeline und sorgt so dafür, dass dieses Objekt in der Pipeline an den nächsten Befehl weitergereicht wird.

Dieser nächste Befehl lautet *Format-Table*. Er gibt die angegebenen Eigenschaften *ServiceName* und *CommandLine* tabellarisch aus.

Dienste-Informationen als CSV in Excel importieren

Sie möchten eine Liste aller laufenden Dienste im kommaseparieren Format exportieren, um die Liste anschließend zum Beispiel in Microsoft Excel weiterzubearbeiten.

Lösung

Verwenden Sie *Export-Csv*, um die Liste der Dienste in einer kommaseparierten Datei zu speichern. Diese Datei kann anschließend von Programmen importiert werden, die kommaseparierte Dateien unterstützen (beispielsweise Microsoft Excel):

```
PS > Get-Service | Where-Object {$_.Status -eq "running"} | →
    Export-Csv liste.csv -Encoding UTF8 -useCulture -NoTypeInfo
PS > notepad liste.csv
PS > .\liste.csv
```

Hintergrund

Das Cmdlet *Export-Csv* exportiert die Objekte in eine kommaseparierte Liste. Dabei werden sämtliche Eigenschaften der Objekte in Text umgewandelt. Das Ergebnis kann deshalb sehr umfangreich sein. Interessieren Sie sich nur für bestimmte Objekteigenschaften, beschränken Sie die Ausgabe mit *Select-Object* auf die relevanten Eigenschaften, bevor Sie die Objekte in eine kommaseparierte Datei exportieren:

```
PS > Get-Service | Where-Object {$_.status -eq "running"} | →
    Select-Object DisplayName, ServiceName, Status | →
    Export-Csv liste.csv -Encoding UTF8 -useCulture -NoTypeInfo
PS > notepad liste.csv
```

Kommaseparierte Listen können von vielen Anwendungsprogrammen importiert werden, zum Beispiel auch von Microsoft Excel. Allerdings schlägt der Importversuch auf deutschen Systemen

häufig fehlt, weil die deutschen Regionaleinstellungen als Trennzeichen nicht ein Komma vorsehen, sondern ein Semikolon. Indem Sie den Parameter *-useCulture* angeben, verwendet *Export-Csv* das für Ihre Ländereinstellungen maßgebliche Trennzeichen. Mit dem Parameter *-Delimiter* kann alternativ auch ein beliebiges Trennzeichen verabredet werden. Indem Sie das Encoding auf *UTF8* einstellen, werden deutsche Sonderzeichen korrekt angezeigt. Mit *-NoTypeInfo* sorgen Sie schließlich dafür, dass PowerShell den Herkunftsdatentyp nicht mit in die exportierte Datei schreibt.

Dienstzustand überprüfen

Sie möchten den aktuellen Zustand der Dienste gegen eine Schablone prüfen und alarmiert werden, wenn der Dienstzustand von der Schablone abweicht.

Lösung

Speichern Sie den gewünschten Zustand der Dienste mit *Export-Clixml* in einer XML-Datei. Prüfen Sie den aktuellen Zustand der Dienste dann später zu einem beliebigen Zeitpunkt, indem Sie einen weiteren Schnappschuss mit *Export-Clixml* anlegen und beide Schnappschüsse mit *Compare-Object* vergleichen:

```
PS > Get-Service | Select-Object Name, Status | Export-Clixml vorher.xml
PS > # Ändern Sie Diensteeinstellungen
PS > Get-Service | Select-Object Name, Status | Export-Clixml nachher.xml
PS > $vorher = Import-Clixml vorher.xml
PS > $nachher = Import-Clixml nachher.xml
PS > Compare-Object $vorher $nachher -property Name, Status
```

Name	Status	SideIndicator
----	-----	-----
WSCSV	Stopped	=>
WSCSV	Running	<=

```
PS > Compare-Object $vorher $nachher -property Name, Status | →
? { $_.SideIndicator -eq '>' }
```

Name	Status	SideIndicator
----	-----	-----
WSCSV	Stopped	=>

Hintergrund

Mit *Export-Clixml* werden Objekte in einer Datei »serialisiert«, also in einem Format gespeichert, mit dessen Hilfe sich die Objekte später wieder in Objektform laden lassen. *Import-Clixml* lädt die serialisierten Informationen und konvertiert sie zurück in Objekte.

Damit eignet sich das Gespann *Export-Clixml* und *Import-Clixml* hervorragend dazu, Schnappschüsse aufzunehmen und später mit *Compare-Object* zu vergleichen. Allerdings sind dabei einige wichtige Punkte zu beachten:

- **Serialisierte Daten mit serialisierten Daten vergleichen** Vergleichen Sie nur Daten, die jeweils mit *Export-Clixml* serialisiert wurden und die Sie mit *Import-Clixml* importiert haben. Vergleichen Sie keine Objekte, die Sie mit *Import-Clixml* importiert haben, direkt mit Objekten von *Get-Service*.
- **Eigenschaften festlegen** Legen Sie mit *Select-Object* fest, welche Eigenschaften Sie überhaupt interessieren, und serialisieren Sie nur diese Eigenschaften. Wenn Sie die Objekte später mit *Compare-Object* vergleichen, geben Sie diese Eigenschaften als *-property*-Parameter an.

Neuen Dienst installieren

Sie wollen einen neuen Dienst registrieren, um ihn anschließend starten und verwenden zu können.

Lösung

Verwenden Sie *New-Service*:

```
PS > $cred = Get-Credential
PS > New-Service -name SampleService -binaryPathName →
    "c:\..\WindowsService1.exe" -displayName 'Neuer Dienst' →
    -StartupType Manual -credential $cred
```

Hintergrund

Herkömmliche Programme werden bei Bedarf vom Anwender gestartet und stehen nur diesem Anwender zur Verfügung. Dienste sind Programme, die nicht an einen bestimmten Anwender gebunden sind, sondern vom Windows-System gestartet werden. Wie ihr Name andeutet, stellen Dienste grundlegende Dienstleistungen zur Verfügung.

Dienste liegen, ähnlich wie reguläre Programme, als *.exe*-Datei vor. Damit Sie einen Dienst starten und verwenden können, muss er zunächst registriert werden. Bei dieser Registrierung legen Sie zum Beispiel fest, wie der Dienst gestartet wird (automatisch oder manuell) und unter welchem Benutzerkonto er ausgeführt wird. Normalerweise wird diese Registrierung automatisch vorgenommen, wenn Sie einen Dienst installieren. Mit *New-Service* können Sie dies aber auch manuell vornehmen und anschließend den neu installierten Dienst starten.

Dienst entfernen

Sie möchten einen installierten Dienst entfernen.

Lösung

Verwenden Sie die *Delete()*-Methode des *Win32_Service*-Objekts, um den Dienst zu entfernen.

ACHTUNG Sie entfernen damit die Registrierung des Diensts. Sie können den entfernten Dienst nicht mehr verwenden und müssen ihn gegebenenfalls neu installieren. Entfernen Sie Dienste deshalb nicht leichtfertig! Um einen Dienst zu entfernen, benötigen Sie Administratorrechte. Ist der Dienst (oder Abhängigkeiten davon) in Gebrauch, werden die Änderungen erst nach einem Neustart wirksam.

```
PS > $dienst = Get-WmiObject Win32_Service -filter 'Name="Dienstname"'
PS > $dienst.Delete().ReturnValue
```

Hintergrund

PowerShell enthält kein Cmdlet, um einen einmal installierten Dienst wieder zu entfernen. Möchten Sie Dienste entfernen, greifen Sie deshalb mit *Get-WmiObject* auf den WMI-Dienst zu. Hierdurch wird der Dienst unwiderruflich und ohne weitere Sicherheitsabfrage gelöscht. Tun Sie dies nur, wenn Sie sich über die Konsequenzen im Klaren sind. Das Entfernen von Diensten, die für Windows lebenswichtig sind, kann Ihre gesamte Windows-Installation irreparabel beschädigen.

Damit Sie Dienste ausführen können, müssen Dienste registriert werden. Dabei teilen Sie Windows mit, wo sich das zugrunde liegende Dienstprogramm befindet, in wessen Namen es ausgeführt wird und wer den Dienst zu welchem Zeitpunkt startet. Entfernen Sie einen Dienst, wird diese Registrierung entfernt. Das dem Dienst zugrunde liegende Programm wird aber nicht gelöscht. Sie könnten den Dienst also mit *New-Service* erneut registrieren, sofern Sie die dafür notwendigen Einstellungen und Angaben kennen.

Zusammenfassung

Die wichtigsten Funktionalitäten von Diensten lassen sich über Cmdlets aus der *Service*-Familie kontrollieren:

```
PS > Get-Command -noun Service
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Service	Get-Service [-Name] <String[]> [-Comput...
Cmdlet	New-Service	New-Service [-Name] <String> [-BinaryPath...
Cmdlet	Restart-Service	Restart-Service [-Name] <String[]> [-Forc...
Cmdlet	Resume-Service	Resume-Service [-Name] <String[]> [-PassT...
Cmdlet	Set-Service	Set-Service [-Name] <String> [-ComputerNa...
Cmdlet	Start-Service	Start-Service [-Name] <String[]> [-PassTh...
Cmdlet	Stop-Service	Stop-Service [-Name] <String[]> [-Force] ...
Cmdlet	Suspend-Service	Suspend-Service [-Name] <String[]> [-Pass...

Einige dieser Cmdlets sind bedingt remotefähig, denn diese unterstützen den Parameter *-ComputerName* (nicht jedoch den Parameter *-Credential*, sodass Sie sich an anderen Systemen nicht mit einem Benutzerkonto ausweisen können). Volle Remotefähigkeit erhalten diese Cmdlets aber, wenn Sie sie zusammen mit dem PowerShell Remoting einsetzen (siehe Kapitel 18).

Mehr Informationen über Dienste und volle Remotefähigkeit bietet die WMI (Windows Management Instrumentation), über die ebenfalls alle Aspekte eines Diensts verwaltbar sind. Nur die WMI verfügt über die Möglichkeit, einen Dienst dauerhaft zu entfernen.

Möchten Sie den Erfolg Ihrer Änderungen in einer grafischen Oberfläche nachvollziehen, verwenden Sie das Snap-In *Dienste* aus der Microsoft Management Console (MMC):

```
PS > services.msc
```

ACHTUNG Änderungen werden im Snap-In *Dienste* erst nach einer Aktualisierung angezeigt. Schließen und öffnen Sie das Snap-In *Dienste*, wenn Sie sichergehen wollen, dass Änderungen darin reflektiert werden.

Kapitel 11

Ereignisprotokoll

In diesem Kapitel:

Alle Ereignisprotokolle auflisten	336
Prüfen, ob ein Ereignisprotokoll existiert	338
Die neuesten Ereignisse auflisten	338
Ereignisse mit einer bestimmten ID finden	342
Fehlermeldungen im Ereignisprotokoll finden	349
Nach einem Stichwort suchen	350
Ereignisse nach Häufigkeit gruppieren	351
Neue Einträge in ein Ereignisprotokoll schreiben	353
Ein Ereignisprotokoll archivieren	354
EVT-Dateien einlesen	356
Den Inhalt eines Ereignisprotokolls löschen	357
Remotezugriff auf Ereignisprotokolle	357
Ereignisprotokoll konfigurieren	358
Zusammenfassung	361

Ereignisprotokolle zeichnen die unterschiedlichsten Vorgänge auf, die auf einem Computer geschehen und bilden deshalb eine wichtige Grundlage für Überwachungen und Fehlersuche. PowerShell kann mit seinen Cmdlets nicht nur den Inhalt dieser Logbücher analysieren, sondern auch die grundlegenden Einstellungen der Ereignisprotokolle verwalten, neue Protokolle anlegen oder eigene Einträge hinzufügen.

Auch die Sicherungskopien von Ereignislogbüchern, sogenannte *.evt*-Dateien, lassen sich von PowerShell einlesen und nachträglich analysieren.

Alle Ereignisprotokolle auflisten

Sie wollen wissen, welche Ereignisprotokolle es auf Ihrem System gibt.

Lösung

Verwenden Sie *Get-EventLog* mit dem Parameter *-list*. So erfahren Sie, welche Ereignisprotokolle es gibt und welche Einstellungen für diese gelten:

```
PS > Get-EventLog -list
```

Max(K)	Retain	OverflowAction	Entries	Name
-----	-----	-----	-----	-----
32.768	0	OverwriteAsNeeded	174	Anwendung
15.168	0	OverwriteAsNeeded	0	DFS-Replikation
20.480	0	OverwriteAsNeeded	0	Hardware-Ereignisse
512	7	OverwriteOlder	0	Internet Explorer
20.480	0	OverwriteAsNeeded	0	Key Management Service
8.192	0	OverwriteAsNeeded	0	Media Center
16.384	0	OverwriteAsNeeded	12	Microsoft Office Diagnostics
16.384	0	OverwriteAsNeeded	865	Microsoft Office Sessions
20.480	0	OverwriteAsNeeded	47.459	System
15.360	0	OverwriteAsNeeded	18.393	Windows PowerShell

Möchten Sie lediglich die Namen aller verfügbaren Ereignisprotokolle erfahren, fügen Sie den Parameter *-asString* hinzu:

```
PS > Get-EventLog -list -asString
```

```
Application
DFS Replication
HardwareEvents
Internet Explorer
Key Management Service
Media Center
ODiag
OSession
Security
System
Windows PowerShell
```


Neben den Standard-Ereignislogbüchern existieren seit Windows Vista/Server 2008 weitere anwendungsspezifische Logbücher. Während *Get-EventLog* diese neuen Logbücher nicht berücksichtigt, kann *Get-WinEvent* sie alle auflisten:

```
PS > Get-WinEvent -listlog * -ErrorAction SilentlyContinue
```

WARNUNG: Die Spalte LogMode passt nicht in die Anzeige und wurde entfernt.

LogName	MaximumSizeInBytes	RecordCount
-----	-----	-----
Application	31457280	36792
HardwareEvents	20971520	0
Internet Explorer	1052672	0
Key Management Service	20971520	0
Media Center	8388608	443
ODiag	16777216	32
OSession	16777216	1304
System	20971520	55632
Windows PowerShell	20971520	25546
ForwardedEvents	20971520	
(...)		

Eine Liste nur der Logbuchnamen erhalten Sie so:

```
PS > [System.Diagnostics.Eventing.Reader.EventLogSession]::GlobalSession.GetLogNames()
```

Hintergrund

Der Parameter *-List* des Cmdlets *Get-EventLog* liefert einen Überblick über alle vorhandenen Standard-Ereignislogbücher und listet ihre Grundeinstellungen auf. Wie viel Speicherplatz einem Ereignisprotokoll zugeordnet ist, verrät die Eigenschaft *MaximumKilobytes*. Was geschehen soll, wenn dieser Speicherplatz aufgebraucht ist, bestimmt die Eigenschaft *OverflowAction*.

```
PS > Get-EventLog -list | Select-Object Log, Maxim*, Over*
```

Log	MaximumKilobytes	OverflowAction
---	-----	-----
Application	32768	OverwriteAsNeeded
DFS Replication	15168	OverwriteAsNeeded
HardwareEvents	20480	OverwriteAsNeeded
Internet Explorer	512	OverwriteOlder
Key Management Service	20480	OverwriteAsNeeded
Media Center	8192	OverwriteAsNeeded
ODiag	16384	OverwriteAsNeeded
OSession	16384	OverwriteAsNeeded
Security		
System	20480	OverwriteAsNeeded
Windows PowerShell	15360	OverwriteAsNeeded

Dieser Aufruf liest im Grunde nur die Namen der in der Windows-Registrierungsdatenbank hinterlegten Ereignisprotokolle auf, die Sie auch direkt aus der Registrierungsdatenbank erfragen können:

```
PS > dir HKLM:SYSTEM\CurrentControlSet\Services\Eventlog -Name
Application
HardwareEvents
Internet Explorer
Key Management Service
Media Center
ODiag
OSession
Security
System
Windows PowerShell
```

Prüfen, ob ein Ereignisprotokoll existiert

Sie möchten herausfinden, ob es ein bestimmtes Ereignisprotokoll auf dem Computer gibt.

Lösung

Verwenden Sie die statische Methode *Exists()* der Klasse *System.Diagnostics.EventLog*. Die folgende Zeile überprüft, ob es das Ereignisprotokoll namens »Application« gibt:

```
PS > [System.Diagnostics.EventLog]::Exists('Application')
True
```

Sie können auf diese Weise allerdings nur die Standardprotokolle überprüfen, die *Get-EventLog* verwaltet. Anwendungsspezifische Logbücher werden darüber nicht erfasst.

Hintergrund

Ereignisprotokolle fassen Ereignisse zu bestimmten Themen oder Gebieten zusammen. Neben den Standardereignisprotokollen *Application*, *System* und *Security*, die es auf jedem Windows-System gibt, können weitere Ereignisprotokolle vorhanden sein, die von anderen Herstellern oder nachinstallierten Komponenten eingerichtet wurden.

Neueste Ereignisse auflisten

Sie möchten nur die aktuellsten Einträge aus einem Ereignisprotokoll anzeigen.

Lösung

Verwenden Sie *Get-EventLog* mit dem Parameter *-Newest*, und geben Sie an, wie viele Einträge Sie sehen möchten. Die folgende Zeile listet die zehn neuesten Einträge aus dem Systemereignisprotokoll auf:

```
PS > Get-EventLog System -Newest 10
```

Index	Time	Type	Source	EventID	Message
81767	Apr 11 10:01	Info	Service Control M...	7036	Die Beschreibu...
81766	Apr 11 09:45	Info	Service Control M...	7036	Die Beschreibu...
81765	Apr 11 09:44	Info	Service Control M...	7036	Die Beschreibu...
81764	Apr 11 09:28	Info	Service Control M...	7036	Die Beschreibu...

(...)

Damit die Ereignismeldungen nicht abgeschnitten werden, verwenden Sie *Format-Table* mit dem Parameter *-wrap*:

```
PS > Get-EventLog System -newest 3 | Format-Table Source, Message -auto -wrap
```

Source	Message
Service Control Manager	Die Beschreibung für Ereignis-ID 1073748860 in Quelle Service Control Manager wurde nicht gefunden. Der lokale Computer hat möglicherweise nicht die notwendigen Registrierungsinformationen oder Meldungs-DLL-Dateien, um die Meldung anzuzeigen, oder Sie sind nicht berechtigt, darauf zuzugreifen. Die folgenden Informationen sind Teil des Ereignisses : 'WinHTTP-Web Proxy Auto-Discovery-Dienst', 'Beendet'

(...)

Mit den Parametern *-Before* und *-After* lassen sich auch Ereignisse aus einem bestimmten Zeitraum ausgeben. Die folgende Zeile listet alle Error-Ereignisse der vergangenen 24 Stunden aus dem Systemlogbuch auf:

```
PS > $gestern = (Get-Date) - (New-Timespan -day 1)
PS > Get-EventLog System -EntryType Error -After $gestern
```

Möchten Sie anwendungsspezifische Logbücher lesen, setzen Sie *Get-WinEvent* ein. Die Namen der entsprechenden Logbücher finden Sie in der Ereignisanzeige, die sich mit *Show-EventLog* öffnen lässt. Sie finden die Logbücher darin im Knoten *Anwendungs- und Dienstprotokolle*.

Um beispielsweise die letzten drei Änderungen am Softwarebestand zu untersuchen, also Installations- und Deinstallationsvorgänge, schauen Sie ins Logbuch *Microsoft-Windows-Application-Experience/Program-Inventory*:

```
PS > Get-WinEvent Microsoft-Windows-Application-Experience/Program-Inventory -MaxEvents 3 | Select-Object TimeCreated, Message | Format-Table -AutoSize -Wrap
```

TimeCreated	Message
15.02.2011 13:49:31	Eine PDU-Instanz (Program Data Updater) wurde mit den folgenden Informationen ausgeführt: Startzeit: ?2011?-?02?-?15T12:44:04.532319100Z, Endzeit: ?2011?-?02?-?15T12:49:31.196892800Z, Beendigungscode: 224, Anzahl von neuen Programmen: 0, Anzahl von entfernten Programmen: 1, Anzahl von aktualisierten Programmen: 2, Anzahl von installierten Programmen: 133, Anzahl von neuen verwaisten Dateien: 3, Anzahl von neuen Add-Ons: 0, Anzahl von entfernten Add-Ons: 1, Anzahl von aktualisierten Add-Ons: 0, Anzahl von installierten Add-Ons: 97, Anzahl von neuen Installationen: 0
15.02.2011 13:49:28	Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten. Name: Windows Live Essentials Version: 15.4.3502.0922 Herausgeber: Microsoft Corporation
15.02.2011 13:49:28	Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten. Name: Mozilla Firefox (3.6.3) Version: 3.6.3 (de) Herausgeber: Mozilla

Möchten Sie dienst- und anwendungsspezifische Logbucheinträge zeitlich beschränken, übergeben Sie die Filterkriterien als Hashtable und legen das Datum in *StartTime* fest. Die folgende Zeile listet alle Software-Änderungen der letzten 24 Stunden auf:

```
PS > $gestern = (Get-Date) - (New-Timespan -day 1)
PS > Get-WinEvent @{Logname = 'Microsoft-Windows-Application-Experience/Program-Inventory'; StartTime=$gestern} | Format-Table TimeCreated, Message -AutoSize -Wrap
```

TimeCreated	Message
15.02.2011 13:49:31	Eine PDU-Instanz (Program Data Updater) wurde mit den folgenden Informationen ausgeführt: Startzeit: ?2011?-?02?-?15T12:44:04.532319100Z, Endzeit: ?2011?-?02?-?15T12:49:31.196892800Z, Beendigungscode: 224, Anzahl von neuen Programmen: 0, Anzahl von entfernten Programmen: 1, Anzahl von aktualisierten Programmen: 2, Anzahl von installierten Programmen: 133, Anzahl von neuen verwaisten Dateien: 3, Anzahl von neuen Add-Ons: 0, Anzahl von entfernten Add-Ons: 1, Anzahl von aktualisierten Add-Ons: 0, Anzahl von installierten Add-Ons: 97, Anzahl von neuen Installationen: 0

15.02.2011 13:49:28 Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten.

Name: Windows Live Essentials
Version: 15.4.3502.0922
Herausgeber: Microsoft Corporation

15.02.2011 13:49:28 Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten.

Name: Mozilla Firefox (3.6.3)
Version: 3.6.3 (de)
Herausgeber: Mozilla

Hintergrund

Get-EventLog liefert sämtliche Einträge eines klassischen Ereignisprotokolls. Welche Ereignisprotokolle es gibt, finden Sie mit der Lösung »Alle Ereignisprotokolle auflisten« auf Seite 336 heraus. Mit dem Parameter *-Newest* greifen Sie nur auf die neuesten Einträge zu und geben dabei an, wie viele Einträge Sie sehen möchten. Daneben gibt es viele weitere Parameter, die so heißen wie die Spalten des Ergebnisses und mit denen Sie die Ergebnisse nach diesen Spalten filtern.

Möchten Sie beispielsweise nur Ereignisse aus einer bestimmten Ereignisquelle lesen, erkennen Sie im Ergebnis von *Get-EventLog*, dass die Ereignisquelle in der Spalte *Source* zu finden ist. Mit dem gleichnamigen Parameter *-Source* beschränken Sie also die Ergebnisse auf eine bestimmte Quelle.

Eine Besonderheit bilden die Parameter *-Before* und *-After*, die eine Zeitangabe verlangen und dann nur diejenigen Ereignisse liefern, die im angegebenen Zeitintervall geschrieben wurden (Spalte *Time*).

Neben den klassischen Ereignislogbüchern existieren seit Einführung von Windows Vista/Server 2008 zahlreiche dienst- und anwendungsspezifische weitere Logbücher, die detaillierte Informationen über den Zustand und die Konfiguration des Computers liefern. Um diese Informationen abzurufen, setzt man *Get-WinEvent* ein. Hier übernimmt der Parameter *-MaxEvents* die Rolle von *-Newest* und bestimmt, wie viele Ergebnisse Sie sehen möchten. Da *Get-WinEvent* die neuesten Ereignisse zuerst liefert, sehen Sie damit jeweils die aktuellsten Einträge. Sie können die Reihenfolge aber auch mit *-Oldest* umdrehen.

Die Filterung nach bestimmten Ereignissen erfolgt bei *Get-WinEvent* etwas anders als bei *Get-EventLog*. Sie legen dazu ein Hashtable an, in dem Sie die einzelnen Filterkriterien aufführen. Es gibt hier also keine festen Parameter mehr zur Filterung, was sinnvoll ist, weil dienst- und anwendungsspezifische Ereignisse je nach Typ ganz unterschiedliche Eigenschaften enthalten.

Um mit *Get-WinEvent* sinnvoll arbeiten zu können, müssen Sie natürlich zuerst den Namen des jeweiligen Logbuchs kennen. Einen guten Überblick über die verfügbaren Logbücher bietet wie bereits oben beschrieben die Ereignisanzeige, doch sind die Namen der Logbücher darin lokalisiert, also auf einem deutschen System in deutsch. Benötigt werden aber die englischsprachigen Originalnamen. Diesen finden Sie heraus, wenn Sie in einem Logbuch ein Ereignis anklicken

und in der Ergebnisanzeige die Registerkarte *XML* wählen. Der englischsprachige Name des zugrunde liegenden Logbuchs wird in den XML-Daten im Tag *Channel* genannt.

Sie können sich auch am Provider des Logbuchs orientieren. Dieser entspricht dem gelben Ordner in der linken Spalte der Ereignisanzeige und ist nicht lokalisiert, dafür aber gekürzt. Verwenden Sie diesen Namen als Suchbegriff, dann liefert *Get-WinEvent* alle Logbuchnamen dieses Knotens.

Möchten Sie also die Originalnamen aller Logbücher im Knoten *Application-Experience* erfahren, gehen Sie so vor:

```
PS > Get-WinEvent -ListLog *Application-Experience* | →
    Select-Object -ExpandProperty LogName
Microsoft-Windows-Application-Experience/Problem-Steps-Recorder
Microsoft-Windows-Application-Experience/Program-Compatibility-Assistant
Microsoft-Windows-Application-Experience/Program-Compatibility-Troubleshooter
Microsoft-Windows-Application-Experience/Program-Inventory
Microsoft-Windows-Application-Experience/Program-Telemetry
```

Sie können sich auch sämtliche Logbücher anzeigen lassen, die mindestens zehn Einträge enthalten. Eine Liste aller solcher Logbücher, absteigend nach Inhaltsmenge sortiert, liefert diese Zeile:

```
PS > Get-WinEvent -Listlog * -ea 0 | Where-Object { $_.RecordCount -gt 10 } | →
    Sort-Object RecordCount -Descending | Select-Object LogName, RecordCount
```

LogName	RecordCount
-----	-----
System	55647
Application	36792
Windows PowerShell	25474
Microsoft-Windows-PowerShell/Operational	16922
Microsoft-Windows-WinRM/Operational	2554
Microsoft-Windows-NlaSvc/Operational	2451
Microsoft-Windows-WindowsUpdateClient/Operational	2385
Microsoft-Windows-Dhcp-Client/Admin	2370
Microsoft-Windows-OfflineFiles/Operational	2348
Microsoft-Windows-User Profile Service/Operational	2263
Microsoft-Windows-NetworkProfile/Operational	2143
Microsoft-Windows-Diagnosis-DPS/Operational	1857
(...)	

Ereignisse mit einer bestimmten ID finden

Sie möchten aus einem oder mehreren Ereignisprotokollen nur Ereignisse eines bestimmten Typs auflisten, zum Beispiel, um sich schnell über bestimmte Fragestellungen zu informieren.

Lösung

Filtern Sie die Ereignisse, indem Sie die gewünschte Ereignisquelle angeben. Alle Ereignisse rund um den *WindowsUpdateClient* erhalten Sie zum Beispiel so:

```
PS > Get-EventLog System -Source 'Microsoft-Windows-WindowsUpdateClient'
```

Wenn Sie nicht den vollständigen Namen der Ereignisquelle angeben können oder wollen, verwenden Sie Platzhalterzeichen:

```
PS > Get-EventLog System -Source '*WindowsUpdateClient'
```

Jede Ereignisquelle vergibt für unterschiedliche Ereignisse darüber hinaus eine Kategorie, die Ereignis-ID. Sie findet sich in der Eigenschaft *EventID*. Der *WindowsUpdateClient* meldet unter der Ereignis-ID 19 beispielsweise erfolgreich installierte Updates:

```
PS > Get-EventLog system -Source '*WindowsUpdateClient' -InstanceID 19 | Format-Table -wrap TimeGenerated, Message
```

Wollen Sie bestimmte Ereignisse aus einem dienst- oder anwendungsspezifischen Logbuch ermitteln, setzen Sie *Get-WinEvent* ein. Möchten Sie beispielsweise im Logbuch »Microsoft-Windows-Application-Experience/Program-Inventory« alle Ereignisse mit der ID 905 lesen (Ereignisse mit dieser ID-Nummer protokollieren Software-Updates), gehen Sie so vor:

```
PS > Get-WinEvent -FilterHashtable @{logname='Microsoft-Windows-Application-Experience/Program-Inventory'; id=905} | Format-Table TimeCreated, Message -AutoSize -Wrap
```

TimeCreated	Message
-----	-----
15.02.2011 13:49:28	Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten.
	Name: Windows Live Essentials
	Version: 15.4.3502.0922
	Herausgeber: Microsoft Corporation
(...)	

Hintergrund

Ereigniseinträge sind immer einer bestimmten Quelle zugeordnet. Jede Quelle thematisiert Einträge danach mit einer Ereignis-ID. Ein und dieselbe Ereignis-ID kann also ganz unterschiedliche Eintragsarten liefern, wenn man nicht außerdem das Ereignisprotokoll und die Quelle berücksichtigt. Leider stimmen die Angaben, die *Get-EventLog* meldet, nicht immer mit den Angaben überein, die Sie über den Aufruf von *eventvwr.msc* in der Ereignisanzeige sehen.

Zum Beispiel wird der Name der Ereignisquelle in der Ereignisanzeige in der Spalte *Quelle* abgekürzt angezeigt. Auch die Nachricht, die *Get-EventLog* in der Eigenschaft *Message* liefert, ist häufig nicht vollständig. Der Grund: Ereignisse können aus einer vorgegebenen Textmaske bestehen, in die die eigentlichen Ereignisdaten nur noch an vorgegebenen Stellen eingefügt werden. Ist die Maske nicht verfügbar, erscheint die Meldung nur bruchstückhaft.

Allerdings finden Sie die entscheidenden Ereignisinformationen in der Eigenschaft *ReplacementStrings*. Sie enthält ein Feld mit den Textbausteinen, die in die Schablone eingefügt werden. Die scheinbare Einschränkung ist also eigentlich ein Gewinn, weil Sie mithilfe von *ReplacementStrings* direkt und ohne Textparsing an die eigentlichen Ereignisdaten herankommen. Sie müssen nur noch wissen, an welcher Position in diesem Feld die für Sie wichtigen Angaben stehen:

```
PS > Get-EventLog system -Source '*WindowsUpdateClient' -InstanceID 19 | →
    Format-Table -wrap TimeGenerated, ReplacementStrings
```

TimeGenerated	ReplacementStrings
-----	-----
11.02.2011 08:13:14	{Definitionsupdate für Microsoft Security Essentials - KB2310138 (Definition 1.97.14 91.0), {AF0CAB9D-A269-443F-B405-C27DBFA6B3 B4}, 100}
10.02.2011 03:26:36	{Update für Windows 7 (KB2502285), {0BCA6C 00-4FD3-4280-96BE-B89988FA1702}, 101}
(...)	

So wird deutlich, dass das Ereignis mit der ID 19 aus der Quelle *WindowsUpdateClient* drei Informationen in die Schablone einfügt, denn die Eigenschaft *ReplacementStrings* enthält offenbar drei Angaben. Die erste davon ist der Name des installierten Updates. Durch den direkten Zugriff auf die passenden Bausteine in der Eigenschaft *ReplacementStrings* kann man sich also gezielt die gewünschten Informationen ausgeben.

```
PS > Get-EventLog system -Source '*WindowsUpdateClient' -InstanceID 19 | →
    ForEach-Object { '{0}: {1}' -f $_.TimeGenerated, $_.ReplacementStrings[0] }
```

```
10.02.2011 03:26:36: Update für Windows 7 (KB2454826)
10.02.2011 03:26:36: Sicherheitsupdate für Windows 7 (KB2393802)
10.02.2011 03:26:36: Kumulatives Sicherheitsupdate für Internet Explorer
                        8 für Windows 7 (KB2482017)
(...)
```

Auch über die WMI kann man Ereigniseinträge abrufen. Weil dabei die Filterung serverseitig stattfinden kann, ist dieser Weg insbesondere für Remoteabfragen besser geeignet.


```
PS > Get-WmiObject Win32_NTLogEvent -filter 'LogFile="System" and →
      SourceName="Microsoft-Windows-WindowsUpdateClient" and EventCode=19' | →
      Format-Table TimeWritten, Message -wrap
```

TimeWritten	Message
-----	-----
20080115075405.591200-000	Installation erfolgreich: Das folgende Update wurde installiert. Definition Update for Windows Defender - KB915597 (Definition 1.24.5399.0)
(...)	

Allerdings unterscheiden sich die Ereignis-Objekte, die Sie auf diese Weise zurückerhalten. So enthält zum Beispiel die Eigenschaft *TimeWritten* das Ereignisdatum im speziellen WMI-Format und Sie müssten es zuerst in eine lesbare Zeitangabe umwandeln:

```
PS > Get-WmiObject Win32_NTLogEvent -filter 'LogFile="System" and →
      SourceName="Microsoft-Windows-WindowsUpdateClient" and EventCode=19' | →
      % { "{0}: {1}" -f (([wmi]'').ConvertToDateTime($_.TimeWritten)) , $_.Message }
```

```
20.02.2011 09:58:50: Installation erfolgreich: Das folgende Update wurde installiert.
Definitionsupdate für Microsoft Security Essentials - KB2310138 (Definition 1.97.2166.0)
19.02.2011 09:47:53: Installation erfolgreich: Das folgende Update wurde installiert.
Definitionsupdate für Microsoft Security Essentials - KB2310138 (Definition 1.97.2124.0)
(...)
```

Andere Eigenschaften tragen neue Namen. Die Textbausteine für die Ereignisnachricht werden im WMI-Objekt nicht in der Eigenschaft *ReplacementStrings* gespeichert, sondern in der Eigenschaft *InsertionStrings*:

```
PS > Get-WmiObject Win32_NTLogEvent -filter 'LogFile="System" and →
      SourceName="Microsoft-Windows-WindowsUpdateClient" and EventCode=19' | →
      % { '{0}: {1}' -f (([wmi]'').ConvertToDateTime($_.TimeWritten)) , →
      $_.InsertionStrings[0] }
```

```
20.02.2011 09:58:50: Definitionsupdate für Microsoft Security Essentials - KB2310138
(Definition 1.97.2166.0)
19.02.2011 09:47:53: Definitionsupdate für Microsoft Security Essentials - KB2310138
(Definition 1.97.2124.0)
(...)
```

Auch bei den dienst- und anwendungsspezifischen Protokollen, die Sie über *Get-WinEvent* abrufen, stehen die in den Meldungen enthaltenen Informationen in der Eigenschaft *Properties* des jeweiligen Ereignis-Objekts zur Verfügung. Die folgende Zeile ruft das erste Ereignis mit der ID 905 ab, die für Software-Aktualisierungen steht, und greift auf dessen Eigenschaft *Properties* zu. Ausgegeben werden die Informationen über die erfolgte Software-Aktualisierung, die normalerweise in die Textschablone der Meldung eingebettet würden:

```
PS > (Get-WinEvent -MaxEvents 1 -FilterHashtable @{
    @{'logname='Microsoft-Windows-Application-Experience/Program-Inventory';
    id=905}}).Properties
```

Value

```
Windows Live Essentials
15.4.3502.0922
Microsoft Corporation
7
AddRemoveProgram
0000f70036ca03cf2572b77565d2fee62adc00000700
0000c731edd1bb2130a050ce28e5adaa29b1db3d94e3
00003f571b697468000bab86a161c70e01eeb435ce03
```

Die zugehörige Meldung finden Sie in der Eigenschaft *Message*:

```
PS > (Get-WinEvent -MaxEvents 1 -FilterHashtable @{
    @{'logname='Microsoft-Windows-Application-Experience/Program-Inventory';
    id=905}}).Message
```

Im System wurde ein Programm aktualisiert. Details finden Sie in den Ereignisdaten.

```
Name: Windows Live Essentials
Version: 15.4.3502.0922
Herausgeber: Microsoft Corporation
```

Name, Version und Herausgeber der aktualisierten Software finden sich also in *Properties* an erster, zweiter und dritter Stelle. So ließe sich eine Funktion wie beispielsweise *Get-SoftwareUpdates* erstellen, die einen Report über erfolgte Software-Updates erzeugt:

```
function Get-SoftwareUpdates {
    $filter = @{
        logname='Microsoft-Windows-Application-Experience/Program-Inventory'
        id=905
    }

    Get-WinEvent -FilterHashtable $filter |
    ForEach-Object {
        $info = 1 | Select-Object Datum, Anwendung, Version, Herausgeber
        $info.Datum = $_.TimeCreated
        $info.Anwendung = $_.Properties[0].Value
        $info.Version = $_.Properties[1].Value
        $info.Herausgeber = $_.Properties[2].Value
        $info
    }
}
```

```
PS > Get-SoftwareUpdates
```

Datum	Anwendung	Version	Herausgeber
-----	-----	-----	-----
15.02.2011 13:49:28	Windows Live...	15.4.3502.0922	Microsof...
15.02.2011 13:49:28	Mozilla Fire...	3.6.3 (de)	Mozilla
13.02.2011 11:28:17	Windows Live...	15.4.3502.0922	Microsof...
(...)			

Nach gleichem Muster lassen sich so die äußerst wertvollen Informationen der dienst- und anwendungsspezifischen Logbücher bergen und für Analysen bereitstellen. Das Logbuch »Microsoft-Windows-Diagnostics-Performance/Operational« protokolliert beispielsweise sämtliche Startvorgänge und meldet ungewöhnliche Verzögerungen.

Um die nützlichen Informationen in diesen Ereignissen zu identifizieren, kann man das Ergebnis an *Out-GridView* weiterleiten.

```
PS > Get-WinEvent Microsoft-Windows-Diagnostics-Performance/Operational | Sort-Object ID | Out-GridView
```

HINWEIS

Out-GridView wird auf Servern nicht standardmäßig installiert. Es steht nur zur Verfügung, wenn PowerShell ISE (Integrated Scripting Environment) installiert ist.

Zum Abruf bestimmter Logbuchinformationen sind Administratorrechte erforderlich. Das gilt auch für das in diesem Beispiel verwendete Logbuch mit den Performedaten.

Wie sich zeigt, repräsentiert die ID 100 den Windows-Start, und in der Meldung entspricht die erste Information der Startdauer in Millisekunden, die zweite Information meldet, ob es ungewöhnliche Beeinträchtigungen gab, und die dritte Information zeigt das Datum des Starts an.

Um zu erfahren, wo diese Informationen in der Eigenschaft *Properties* definiert sind, schaut man sich das Ereignis in der Ereignisanzeige (*Show-EventLog*) in seiner XML-Darstellung an. Darin ist die Zuordnung im Knoten *EventData* erkennbar:

```
- <EventData>
  <Data Name="BootTsVersion">2</Data>
  <Data Name="BootStartTime">2011-02-14T19:44:12.702800400Z</Data>
  <Data Name="BootEndTime">2011-02-14T19:46:39.363092900Z</Data>
  <Data Name="SystemBootInstance">121</Data>
  <Data Name="UserBootInstance">91</Data>
  <Data Name="BootTime">103151</Data>
  <Data Name="MainPathBootTime">38051</Data>
  <Data Name="BootKernelInitTime">22</Data>
  <Data Name="BootDriverInitTime">222</Data>
  <Data Name="BootDevicesInitTime">2609</Data>
  <Data Name="BootPrefetchInitTime">0</Data>
  <Data Name="BootPrefetchBytes">0</Data>
  <Data Name="BootAutoChkTime">0</Data>
  <Data Name="BootSmssInitTime">13460</Data>
```

```

<Data Name="BootCriticalServicesInitTime">1169</Data>
<Data Name="BootUserProfileProcessingTime">6001</Data>
<Data Name="BootMachineProfileProcessingTime">314</Data>
<Data Name="BootExplorerInitTime">2340</Data>
<Data Name="BootNumStartupApps">15</Data>
<Data Name="BootPostBootTime">65100</Data>
<Data Name="BootIsRebootAfterInstall">false</Data>
<Data Name="BootRootCauseStepImprovementBits">4194304</Data>
<Data Name="BootRootCauseGradualImprovementBits">0</Data>
<Data Name="BootRootCauseStepDegradationBits">0</Data>
<Data Name="BootRootCauseGradualDegradationBits">0</Data>
<Data Name="BootIsDegradation">false</Data>
<Data Name="BootIsStepDegradation">false</Data>
<Data Name="BootIsGradualDegradation">false</Data>
<Data Name="BootImprovementDelta">0</Data>
<Data Name="BootDegradationDelta">0</Data>
<Data Name="BootIsRootCauseIdentified">true</Data>
<Data Name="OSLoaderDuration">2939</Data>
<Data Name="BootPNPInitStartTimeMS">22</Data>
<Data Name="BootPNPInitDuration">2632</Data>
<Data Name="OtherKernelInitDuration">3084</Data>
<Data Name="SystemPNPInitStartTimeMS">5664</Data>
<Data Name="SystemPNPInitDuration">199</Data>
<Data Name="SessionInitStartTimeMS">5916</Data>
<Data Name="Session0InitDuration">4302</Data>
<Data Name="Session1InitDuration">2078</Data>
<Data Name="SessionInitOtherDuration">7078</Data>
<Data Name="WinLogonStartTimeMS">19376</Data>
<Data Name="OtherLogonInitActivityDuration">10019</Data>
<Data Name="UserLogonWaitDuration">20591</Data>
</EventData>

```

Nachdem nun klar ist, an welchen Positionen innerhalb der *Properties*-Eigenschaft die gewünschten Informationen zu finden sind, kann eine passende Funktion nach demselben Strickmuster wie eben erzeugt werden:

```

function Get-WindowsLaunch {
    $filter = @{
        logname='Microsoft-Windows-Diagnostics-Performance/Operational'
        id=100
    }

    Get-WinEvent -FilterHashtable $filter |
    ForEach-Object {
        $info = 1 | Select-Object Datum, Startdauer, Startprogramme, Logonzeit
        $info.Datum = $_.Properties[1].Value
        $info.Startdauer = $_.Properties[5].Value
        $info.Startprogramme = $_.Properties[18].Value
        $info.Logonzeit = $_.Properties[43].Value
        $info
    }
}

```

```
PS > Get-WindowsLaunch
```

Datum	Startdauer	Startprogramme	Logonzeit
-----	-----	-----	-----
14.02.2011 20:44:12	103151	15	20591
12.02.2011 23:47:58	108445	15	134423
11.02.2011 17:53:44	130913	15	7375
11.02.2011 06:50:25	199194	18	21031
10.02.2011 17:26:39	95689	15	10317
07.02.2011 16:56:58	131289	16	8854
05.02.2011 09:07:24	129181	15	14743
03.02.2011 08:04:20	74992	12	30398
30.01.2011 16:06:31	96325	16	37080
28.01.2011 17:17:15	134119	15	7946797
(...)			

Über *Measure-Object* könnten Sie nun detaillierte Analysen zum Windows-Startverhalten durchführen. Die durchschnittliche Startzeit sowie Minimal- und Maximalwerte erhielten Sie jetzt so:

```
PS > Get-WindowsLaunch | Measure-Object -property Startdauer -average -min -max
```

```
Count      : 29
Average    : 130326,586206897
Sum        :
Maximum    : 199194
Minimum    : 68191
Property   : Startdauer
```

Fehlermeldungen im Ereignisprotokoll finden

Sie möchten aus einem Ereignisprotokoll nur Fehlermeldungen auslesen und weniger wichtige Ereignisse überspringen.

Lösung

Filtern Sie die Ereignisse mithilfe der Eigenschaft *-EntryType*. Die folgende Zeile liefert nur *Error*-Ereignisse aus dem Systemereignisprotokoll:

```
PS > Get-EventLog System -EntryType 'Error'
```

In dienst- und anwendungsspezifischen Logbüchern wird die Art des Eintrags in den Eigenschaften *Level* und *LevelDisplayName* geführt. Möchten Sie beispielsweise alle Einträge zu Windows Backup lesen, lautet das dafür zuständige Logbuch »Microsoft-Windows-Backup«:

```
PS > Get-WinEvent Microsoft-Windows-Backup | →
      Format-Table TimeCreated, LevelDisplayName, →
      Level, Message -AutoSize
```

TimeCreated	LevelDisplayName	Level	Message
06.02.2011 20:18:47	Informationen	4	Die Sicherung ist abgeschl...
06.02.2011 20:18:47	Fehler	2	Die Sicherung wurde abgebr...
06.02.2011 19:28:51	Informationen	4	Die Sicherung wurde gestar...
05.02.2011 14:39:02	Informationen	4	Die Sicherung ist abgeschl...
05.02.2011 14:39:02	Informationen	4	Die Sicherung wurde erfolg...
05.02.2011 11:46:53	Informationen	4	Die Sicherung wurde gestar...

Möchten Sie nur Fehler sehen, filtern Sie nach der Eigenschaft *Level*:

```
PS > Get-WinEvent -FilterHashTable @{LogName = 'Microsoft-Windows-Backup'; →
      Level=2}
```

TimeCreated	ProviderName	Id	Message
06.02.2011 20:18:47	Microsoft-Windows-Backup	8	Die Sicherung...

Hintergrund

Ereignisse werden je nach Schweregrad in die drei Typen *Information*, *Warnung* und *Fehler* eingestuft. In der Eigenschaft *EntryType* findet sich entsprechend die Angabe *Information*, *Warning* und *Error*. Bei dienst- und anwendungsspezifischen Logbüchern findet sich diese Information als Zahlenwert in der Eigenschaft *Level* und als Klartextname in der Eigenschaft *LevelDisplayName*:

- **Information** Ereignisse der Kategorie »Information« können ignoriert werden. Solche Ereignisse protokollieren lediglich bestimmte Vorgänge aus Gründen der späteren Nachvollziehbarkeit.
- **Warnung** Ereignisse der Kategorie »Warning« können wichtige Informationen über drohende Ausfälle oder sonstige ungewöhnliche Vorkommnisse enthalten. Hier wird zum Beispiel gemeldet, wenn die Systemressourcen zur Neige gehen oder bestimmte Vorgänge verdächtig lange dauern. Sie sollten im Rahmen der Systempflege sorgfältig untersucht werden, um Ausfallzeiten und Datenverlust vorzubeugen.
- **Fehler** Ereignisse der Kategorie »Error« sind ernsthafte Fehler, die bei einem einwandfreien System nicht auftreten sollten. Solche Einträge müssen zwar nicht zwingend Konsequenzen haben, doch sollten Sie Ereignisse dieser Kategorie auf jeden Fall beachten und die Ursachen ermitteln.

Nach einem Stichwort suchen

Sie möchten alle Ereignisseinträge finden, die ein bestimmtes Stichwort enthalten.

Lösung

Rufen Sie die Ereignisse des gewünschten Ereignisprotokolls ab, und filtern Sie die Ereignisse, indem Sie das Suchwort im Nachrichtentext der Ereignisse suchen. Die folgende Zeile findet alle Ereignisse im Systemereignisprotokoll, die in ihrer Nachricht das Wort »Defender« enthalten:

```
PS > Get-EventLog System | Where-Object { $_.Message -like '*Defender*' }
```

Index	Time	Type	Source	EventID	Message
81687	Apr 10 01:54	Info	WinDefend	1001	Die %Windows-Defender-...
81684	Apr 10 01:47	Info	Microsoft...	19	Die Beschreibung für E...
(...)					

Hintergrund

Die Eigenschaft *Message* enthält den jeweiligen Nachrichtentext eines Ereignisses. Mit den Operatoren *-like* und *-match* können Sie darin ein Textmuster suchen. Wird es gefunden, lässt *Where-Object* (Kurzform »?«) dieses Ereignis durch die Pipeline hindurch. Alle Ereignisse, die das Suchwort nicht enthalten, werden herausgefiltert. Während Sie bei *-like* die im Dateisystem üblichen Platzhalterzeichen verwenden, erwartet *-match* einen regulären Ausdruck.

Ereignisse nach Häufigkeit gruppieren

Sie möchten wissen, wie häufig bestimmte Ereignisse vorkommen.

Lösung

Rufen Sie die Ereignisse mit *Get-EventLog* ab und leiten Sie sie dann an *Group-Object* weiter. Geben Sie *Group-Object* dabei die Eigenschaft an, nach der gruppiert werden soll. Möchten Sie zum Beispiel wissen, welche Ereignisquelle wie häufig Ereignisse meldet, wählen Sie die Eigenschaft *Source*:

```
PS > Get-EventLog System | Group-Object Source -noElement | Sort-Object Count -descending
```

Count	Name
14254	disk
12565	Microsoft-Windows-Serv...
6580	cdrom
3958	Service Control Manager
2540	Tcpip
(...)	

Möchten Sie die Informationen der automatischen Windows-Performance-Diagnose nach Ereignis-ID gruppieren, gehen Sie so vor:

```
PS > Get-WinEvent Microsoft-Windows-Diagnostics-Performance/Operational | →
    Group-Object ID -noElement | Sort-Object Count -descending
```

Hintergrund

Group-Object zählt gleichartige Objekte. Dazu geben Sie *Group-Object* lediglich die Eigenschaft an, anhand derer Sie die Objekte voneinander unterscheiden wollen. Mit dem Parameter *-noElement* verzichten Sie darauf, dass sich *Group-Object* die gruppierten Elemente merkt – eine sinnvolle Wahl, wenn es Ihnen nur auf die Häufigkeitsverteilung ankommt.

Das Ergebnis von *Group-Object* kann anschließend mit *Sort-Object* noch nach Häufigkeit absteigend sortiert werden, damit Sie die häufigsten Ereignisquellen zuerst sehen.

Mit *Group-Object* sind die verschiedensten Analysen möglich. Wollen Sie zum Beispiel herausfinden, welche Ereignisse konkret am häufigsten aufgetreten sind, gruppieren Sie nach der Nachricht, also der Eigenschaft *Message*. Dieser Vorgang kann allerdings relativ lange dauern, weil die Ereignisnachricht oft umfangreiche Datensätze enthält:

```
PS > Get-EventLog System | Group-Object Message -noElement | →
    Sort-Object Count -descending
```

Sie können auch nach mehreren Kriterien gruppieren. Gruppieren Sie zum Beispiel zuerst nach der Quelle und dann nach der *EventID*. So erhalten Sie eine Aufstellung der Häufigkeit bestimmter Ereignis-IDs, gruppiert nach Ereignisquelle:

```
PS > Get-EventLog System | Group-Object Source, EventID -noElement | →
    Sort-Object Count -descending
```

```
Count Name
-----
14252 disk, 51
 6573 cdrom, 51
 3882 Service Control Manage...
 2632 Microsoft-Windows-Serv...
 2485 Tcpip, 4201
(...)
```

Interessieren Sie sich ohnehin nur für eine bestimmte Ereignisquelle, können Sie alle übrigen natürlich zuerst mit *Where-Object* (Kurzform »?«) herausfiltern, was die Gruppierung zudem beschleunigt:

```
PS > Get-EventLog System | Where-Object { $_.Source -like →
    '*WindowsUpdateClient' } | Group-Object EventID -noElement | →
    Sort-Object Count -descending
```


Daraus lassen sich Berichte erstellen:

```
PS > Get-EventLog System | Where-Object { $_.Source -like '*WindowsUpdateClient' } | Group-Object EventID -noElement | Sort-Object Count -descending | % { 'Ereignis-ID {0} trat {1} Mal auf.' -f $_.Name, $_.Count }
```

Ereignis-ID 19 trat 56 Mal auf.
 Ereignis-ID 18 trat 52 Mal auf.
 Ereignis-ID 27 trat 19 Mal auf.
 (...)

Neue Einträge in ein Ereignisprotokoll schreiben

Sie möchten eigene Einträge in ein Ereignisprotokoll schreiben, zum Beispiel, um den Erfolg oder Misserfolg eines Skripts im offiziellen Protokoll des Systems zu vermerken.

Lösung

Um einen Eintrag in ein Ereignisprotokoll zu schreiben, benötigen Sie eine Ereignisquelle. Zum Anlegen einer Ereignisquelle sind Administratorrechte nötig. Eine Ereignisquelle muss nur einmal angelegt werden und steht dann künftig immer unter dem vereinbarten Namen zur Verfügung. Eine Ereignisquelle muss eindeutig sein und darf nicht bereits einem anderen Ereignislogbuch zugeordnet worden sein.

So legen Sie eine neue Ereignisquelle namens *PowerShell Skripts* im Ereignisprotokoll »Anwendungen« an:

```
PS > New-EventLog Application -Source 'PowerShell Skripts'
```

Um einen Eintrag in das Anwendungen-Ereignisprotokoll zu schreiben, verwenden Sie das Cmdlet *Write-EventLog*:

```
PS > Write-EventLog Application -Source 'PowerShell Skripts' -EntryType Information -EventId 12345 -Message 'My own event log entry'
PS > Get-EventLog Application -Source 'PowerShell Skripts'
```

Index	Time	EntryType	Source	InstanceId	Message
----	----	-----	-----	-----	-----
131461	Jan 13 13:36	Information	PowerShell Skripts	12345	My own ...

Das Ergebnis können Sie sich anschließend auch in der Ereignisanzeige anzeigen lassen:

```
PS > Show-EventLog
```

Hintergrund

Ereignisse sind stets einem bestimmten Ereignisprotokoll und einer bestimmten Ereignisquelle zugeordnet. Welche Ereignisquellen es in einem Ereignisprotokoll bereits gibt, finden Sie so heraus:

```
PS > Get-EventLog Application | Group-Object Source -NoElement | Select-Object -ExpandProperty Name | Sort-Object | ForEach-Object { $_.Name }

Application Error
Customer Experience Improvement Program
Desktop Window Manager
ESENT
EventSystem
EvtntAgnt
(...)
```

Diese Liste enthält nur Ereignisquellen, die bereits im Ereignislogbuch vorkommen. Eine komplette Liste aller registrierten Ereignisquellen können Sie sich aus der Registrierungsdatenbank abrufen:

```
PS > $key = 'HKLM:SYSTEM\CurrentControlSet\Services\Eventlog\Application'
PS > dir $key | Select-Object -expandProperty PSChildName | Sort-Object

PSChildName
-----
.NET Runtime
.NET Runtime Optimization Service
Application
(...)
```

Es ist nicht möglich, einzelne Einträge aus einem Ereignisprotokoll zu löschen. Sie können lediglich alle Einträge insgesamt aus einem Ereignisprotokoll löschen.

Möchten Sie eine Ereignisquelle aus einem Ereignislogbuch entfernen, benötigen Sie Administratorrechte und das Cmdlet *Remove-EventLog*:

```
PS > Remove-Eventlog -Source 'PowerShell Skripts'
```

Ereignisprotokoll archivieren

Sie möchten den Inhalt eines Ereignisprotokolls sichern, um die darin enthaltenen Informationen nicht zu verlieren.

Lösung

Ein echtes Backup im sogenannten *.evt*-Format ist nur über WMI und die Methode *Backup-EventLog()* möglich. Die Zieldatei, die Sie angeben, darf noch nicht existieren, und alle angege-

benen Ordner müssen bereits vorhanden sein. Die resultierende Sicherungsdatei kann später von der Ereignisanzeige oder vom Cmdlet *Get-WinEvent* geladen und angezeigt werden. So legen Sie ein *evt*-Backup des Logbuchs *System* in die Datei *c:\sicherung\backup_system.evt* an:

```
PS > md c:\sicherung -ErrorAction SilentlyContinue | Out-Null
PS > $logfile = Get-WmiObject Win32_NTEventLogFile -filter 'LogFileName="System"'
PS > $logfile.BackupEventLog("c:\sicherung\backup_system.evt").returnvalue
0
PS > dir c:\sicherung\*.evt
```

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\sicherung

Mode	LastWriteTime	Length	Name
-a---	11.04.2008 11:22	21041152	backup_system.evt

Wurde das Backup erfolgreich ausgeführt, liefert die Methode den Rückgabewert 0 zurück. Der Rückgabewert 3 bedeutet, dass der angegebene Pfad nicht gefunden wurde (Sie haben möglicherweise einen Ordner angegeben, den es noch nicht gibt). Ein Rückgabewert 80 zeigt an, dass die Sicherungsdatei bereits vorhanden war, und ein Rückgabewert 5 steht für nicht ausreichende Berechtigungen.

Sie können den Inhalt eines Ereignisprotokolls auch mit *Export-Clixml* im XML-Format »serialisieren«, also in einer Datei speichern:

```
PS > Get-EventLog System | Export-Clixml c:\sicherung\backup_system.xml
PS > dir c:\sicherung\back*.xml
```

Die darin gesicherten Einträge erhalten Sie innerhalb von PowerShell folgendermaßen zurück:

```
PS > $entries = Import-Clixml c:\sicherung\backup_system.xml
```

Im Gegensatz zu *.evt*-Dateien, wie sie der WMI-Ansatz liefert, ist der XML-Export zeitaufwändig und resultiert in sehr großen XML-Dateien. Dafür bietet er die Möglichkeit, vor dem Export die Ereignisse zuerst zu filtern und beispielsweise nur bestimmte wichtige Fehlerereignisse zu exportieren.

Hintergrund

Übergeben Sie *BackupEventLog()* als Zielfeldnamen möglichst einen absoluten und nicht einen relativen Pfadnamen an. Der Grund: WMI verwendet nicht denselben aktuellen Ordner wie PowerShell. Wenn Sie also innerhalb von PowerShell mit *Set-Location* oder *cd* in einen anderen Ordner gewechselt sind, befindet sich die WMI noch immer im Standardordner, was zu Verwirrung führen kann. Der aktuelle für WMI relevante Ordner wird aus der Klasse *Environment* mit der Eigenschaft *CurrentDirectory* bestimmt:

```
PS > [Environment]::CurrentDirectory
C:\Windows\system32
```

BackupEventLog() kann das Ereignisprotokoll außerdem nur sichern, wenn die Zielfeile noch nicht existiert. Sorgen Sie also entweder dafür, dass die Zielfeile nicht vorhanden ist, oder geben Sie ihr von vornherein einen eindeutigen Namen mit einem Datums- und Zeitstempel:

```
PS > $log = 'System'
PS > $filename = ".\{0}-Log-{1:MMddyy}.evt" -f $log, (Get-Date)
PS > # Nur für relative Pfade wichtig: in absoluten Pfadnamen verwandeln:
PS > [Environment]::CurrentDirectory = (Get-Item (Get-Location)).FullName
PS > $filename = [System.IO.Path]::GetFullPath($filename)
PS > $logfile = Get-WmiObject Win32_NTEventLogFile -filter 'LogFileName="System"'
PS > $logfile.BackupEventLog($filename).returnValue
0
PS > dir *.evt

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias
Mode                LastWriteTime         Length Name
----                -
-a---             11.04.2008          11:24      21041152 System-Log-04110824.evt
```

EVT-Dateien einlesen

Sie haben eine Sicherung eines Ereignislogbuchs als *.evt*-Datei angelegt und möchten den Inhalt dieser Datei analysieren.

Lösung

Das Cmdlet *Get-WinEvent* kann den Inhalt von *.evt*-Dateien lesen, sodass Sie den Inhalt anschließend analysieren können:

```
PS > Get-WinEvent -Path c:\sicherung\backup_system.evt -Oldest
```

Hintergrund

Get-WinEvent kann mit dem Parameter *-Path* Backupdateien der Ereignislogbücher einlesen. Der Parameter akzeptiert einen oder (kommasepariert) mehrere Pfadnamen. Um die kompletten Informationen zu sehen, die in den Ereignisseinträgen gespeichert sind, leiten Sie das Ergebnis weiter an *Select-Object*:

```
PS > Get-WinEvent -Path c:\sicherung\backup_system.evt -Oldest | Select-Object *
```

Sobald Sie sich einen Überblick über die Eigenschaften verschafft haben, können Sie deren Inhalt für Analysen und Filterungen nutzen. Die folgende Zeile listet beispielsweise nur Fehler-Einträge aus der *.evt*-Datei auf:

```
PS > Get-WinEvent -Path c:\sicherung\backup_system.evt -Oldest -ErrorAction →  
SilentlyContinue | Where-Object { $_.Level -eq 2 }
```

Hierbei ist es nötig, den Parameter *-ErrorAction* anzugeben, damit Fehler unterdrückt werden, die entstehen können, wenn Einträge nicht über die erwarteten Eigenschaften verfügen.

Inhalt eines Ereignisprotokolls löschen

Sie wollen alle Einträge aus einem Ereignisprotokoll löschen.

Lösung

Sprechen Sie das gewünschte Ereignisprotokoll an und setzen Sie *Clear-EventLog* ein. Die folgende Zeile löscht alle Einträge aus dem Anwendungen-Ereignisprotokoll:

```
PS > Clear-EventLog Application
```

Sie benötigen Administratorrechte, um Ereignisprotokollinhalte zu löschen. Für einige Logbücher, wie zum Beispiel das Ereignisprotokoll *Sicherheit*, müssen die Sicherheitsrechte des Benutzerkontos aktiviert sein.

Möchten Sie eines der neuen dienst- und anwendungsspezifischen Logbücher leeren, verwenden Sie diesen Aufruf:

```
PS > [System.Diagnostics.Eventing.Reader.EventLogSession]::GlobalSession →  
.ClearLog("Name des Logbuchs")
```

Hintergrund

Sind Ereignisprotokolle richtig konfiguriert, löschen sie automatisch die jeweils ältesten Einträge, sobald der Platz für neue Einträge knapp wird. Es besteht also selten die Notwendigkeit, Ereignisprotokolle selbst zu löschen. Vor allem gehen dabei alle Informationen aus dem Ereignisprotokoll unwiederbringlich verloren.

Dennoch kann das Löschen sinnvoll sein, wenn Sie zum Beispiel gerade ein Problem behoben haben und nun die folgenden Ereignisse ungestört von älteren Einträgen protokollieren wollen. Sie müssen Ereignisprotokolle manuell in regelmäßigen Abständen löschen, wenn Sie als *OverflowPolicy* die Einstellung *DoNotOverwrite* gewählt haben.

Remotezugriff auf Ereignisprotokolle

Sie möchten auf ein Ereignisprotokoll eines anderen Computers remote über das Netzwerk zugreifen.

Lösung

Das Cmdlet *Get-EventLog* verfügt über den Parameter *-ComputerName*, über den Sie Remote-systeme über DCOM ansprechen können. Allerdings können Sie sich nicht mit einem anderen Benutzerkonto ausweisen.

```
PS > Get-EventLog -LogName System -Newest 10 -EntryType Error -ComputerName PC123
```

Alternativ haben Sie die Möglichkeit, PowerShell Remoting einzusetzen, sofern das Zielsystem ebenfalls PowerShell Version 2 verwendet und Remoting dort eingerichtet ist (siehe Kapitel 18).

```
PS > Invoke-Command { Get-EventLog -LogName System -Newest 10 ->
    -EntryType Error } -ComputerName PC123 -Credential Administrator
```

Über WMI ist ebenfalls ein Remotezugriff auf Ereignislogbücher möglich:

```
PS > Get-WmiObject Win32_NTLogEvent -filter 'LogFile="Application"' ->
    -computer 192.168.2.50 -credential Administrator
```

Hintergrund

Die individuellen Remotefähigkeiten in *Get-Service* und *Get-WmiObject* setzen einige System-konfigurationen voraus. So muss sichergestellt sein, dass die Firewall den Zugriff nicht blockiert. Sollte der Remotezugriff nicht auf Anhieb funktionieren, probieren Sie den Remotezugriff zuerst über die grafischen Werkzeuge wie die Ereignisanzeige aus. So erkennen Sie, ob der Fernzugriff überhaupt möglich ist.

Setzen Sie mit *Invoke-Command* das universelle PowerShell Remoting ein, muss dieses auf dem Zielsystem aktiv sein. Ist PowerShell Remoting einmal korrekt eingerichtet, können sämtliche Befehle damit remotefähig gemacht werden.

Ereignisprotokoll konfigurieren

Sie möchten die Grundeinstellungen eines Ereignisprotokolls kontrollieren oder ändern und zum Beispiel festlegen, was passieren soll, wenn die Maximalgröße des Ereignisprotokolls erreicht ist.

Lösung

Um die aktuellen Einstellungen eines Ereignislogbuchs sichtbar zu machen, verwenden Sie diese Zeile:

```
PS > [System.Diagnostics.Eventlog]'Application' | Select-Object *
```

```
Entries           : {demo5, demo5, demo5, demo5...}
LogDisplayName     : Anwendung
Log               : Application
MachineName       : .
MaximumKilobytes   : 20480
OverflowAction     : OverwriteAsNeeded
MinimumRetentionDays : 0
EnableRaisingEvents : False
SynchronizingObject :
Source            :
Site              :
Container         :
```

Um die Einstellungen zu ändern, sind Administratorrechte erforderlich. Setzen Sie das Cmdlet *Limit-EventLog* ein. Mit dem Parameter *-MaximumSize* wird die Maximalgröße des Logbuchs gesetzt:

```
PS > Limit-EventLog -LogName Application -MaximumSize 30MB
```

Der Parameter *-RetentionDays* bestimmt, wie viele Tage Ereignisse mindestens aufgehoben werden sollen, und *-OverflowAction* legt fest, was geschehen soll, wenn der maximale Speicherplatz erreicht ist. Mögliche Werte hierfür sind *OverwriteAsNeeded*, *OverwriteOlder* oder *DoNotOverwrite*.

Verwenden Sie die Methode *ModifyOverflowPolicy()*, um festzulegen, was passieren soll, wenn die Maximalgröße erreicht ist. Die folgende Zeile legt fest, dass bei Platzmangel die ältesten Einträge zuerst überschrieben werden und dass Ereignisse mindestens einen Tag im Ereignisprotokoll verfügbar sein müssen:

Hintergrund

Ereignisprotokolle unterliegen einerseits gewissen Speicherplatzbeschränkungen und dürfen nicht beliebig viel Speicherplatz konsumieren. Andererseits sind sie ein wichtiger Sicherheitsbaustein, sodass sichergestellt sein muss, dass Ereignisse auch tatsächlich für eine gewisse Dauer protokolliert sind. Ist die maximale Größe des Ereignisprotokolls erreicht, sind drei verschiedene Vorgehensweisen möglich:

Einstellung	Beschreibung
<i>OverwriteAsNeeded</i>	Erreicht das Ereignisprotokoll seine Maximalgröße, werden die ältesten Einträge gelöscht, um Platz für neue Einträge zu schaffen. Dies ist die sicherste Einstellung, denn so ist gewährleistet, dass das Ereignisprotokoll immer neue Einträge aufnehmen kann.
<i>OverwriteOlder</i>	Einträge werden bei Platzmangel nur gelöscht, wenn sie älter sind als in der Eigenschaft <i>MinimumRetentionDays</i> angegeben. Andernfalls können keine neuen Einträge aufgenommen werden.

Tabelle 11.1 Vorgehensweisen, wenn ein Ereignisprotokoll seine Speichergrenze erreicht

Einstellung	Beschreibung
<i>DoNotOverwrite</i>	Ist das Ereignisprotokoll voll, können keine neuen Ereignisse aufgenommen werden, denn es werden keine vorhandenen Ereignisse automatisch gelöscht. Diese Einstellung setzt voraus, dass Sie den Inhalt des Ereignisprotokolls selbst manuell in regelmäßigen Intervallen löschen.

Tabelle 11.1 Vorgehensweisen, wenn ein Ereignisprotokoll seine Speichergrenze erreicht (*Fortsetzung*)

Die Eigenschaft *MinimumRetentionDays* spielt also nur bei der Einstellung *OverwriteOlder* eine Rolle.

Allerdings entsprechen die Einstellungen in der grafischen Oberfläche nicht ganz den programmatischen Einstellmöglichkeiten:

Grafische Oberfläche	Einstellung
<i>Ereignisse bei Bedarf überschreiben</i>	<i>OverwriteOlder</i> oder <i>OverwriteAsNeeded</i> , die Einstellung der <i>MinimumRetentionDays</i> -Eigenschaft und die Auswahl des Überschreibverhaltens ist nicht möglich
<i>Volles Protokoll archivieren, Ereignisse nicht überschreiben</i>	<i>DoNotOverwrite</i> , Protokoll wird automatisch regelmäßig archiviert und gelöscht
<i>Ereignisse nicht überschreiben</i>	<i>DoNotOverwrite</i>

Tabelle 11.2 Zuordnung der Einstellungen zu den Optionen der grafischen Oberfläche

TIPP

Alle Konfigurationsdaten, die Sie hier festlegen, werden in der Windows-Registrierungsdatenbank in diesem Schlüssel hinterlegt:

HKLM:SYSTEM\CurrentControlSet\Services\Eventlog\[Ereignisprotokollname]

Sie können die Einstellungen also auch aus der Registrierungsdatenbank lesen und für das *Application*-Ereignisprotokoll beispielsweise so anzeigen lassen:

```
PS > Get-ItemProperty HKLM:SYSTEM\CurrentControlSet\Services\Eventlog\Application

PSPath           : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\
                  CurrentControlSet\Services\Eventlog\Application
PSParentPath      : Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog
PSCildName        : Application
PSDrive           : HKLM
PSProvider        : Microsoft.PowerShell.Core\Registry
DisplayNameFile   : C:\Windows\system32\wevtapi.dll
DisplayNameID     : 256
PrimaryModule     : Application
File              : C:\Windows\system32\winevt\Logs\Application.evtx
MaxSize           : 33554432
Retention         : 86400
RestrictGuestAccess : 1
Sources           : {MSDMine}
AutoBackupLogFiles : 0
```


Hier wird deutlich, wie die Einstellungen tatsächlich gespeichert werden, denn unabhängig davon, ob Sie die Methode *ModifyOverwritePolicy()* oder die grafische Oberfläche verwenden, spielen nur zwei Registrierungswerte eine Rolle:

- **AutoBackupLogFiles** Wenn 1, werden Ereignisprotokolle automatisch archiviert
- **Retention** Wenn 0, gilt *OverwriteAsNeeded*. Eine positive Zahl aktiviert *OverwriteOlder* und gibt gleichzeitig den Wert für *MinimumRetentionDays* an. Wenn -1, gilt *DoNotOverwrite*.

Sofern Sie über Administratorrechte verfügen, können Sie die Einstellungen für ein Ereignisprotokoll auf diese Weise also auch direkt in der Registrierungsdatenbank ändern. Wollen Sie zum Beispiel dafür sorgen, dass das *Application*-Ereignisprotokoll automatisch gesichert wird (entspricht der Option »Volles Protokoll archivieren, Ereignisse nicht überschreiben«), gehen Sie so vor:

```
PS > Set-ItemProperty HKLM:\SYSTEM\CurrentControlSet\Services\Eventlog\ →
Application AutoBackupLogFiles 1
```

Möchten Sie die Option »Ereignisse bei Bedarf überschreiben« aktivieren, gehen Sie so vor:

```
PS > Set-ItemProperty HKLM:\SYSTEM\CurrentControlSet\Services\Eventlog\ →
Application Retention 0
```

Geben Sie die Zahl 0 an, entspricht dies *OverwriteAsNeeded*. Jede andere positive Zahl entspricht *OverwriteOlder*, nämlich dem gewünschten Wert für *MinimumRetentionDays*. Möchten Sie die Option *Ereignisse nicht überschreiben* wählen, stellen Sie *Retention* auf -1 ein:

```
PS > Set-ItemProperty HKLM:\SYSTEM\CurrentControlSet\Services\Eventlog\ →
Application Retention -1
```

Zusammenfassung

Die Ereignislogbücher von Windows sind eine exzellente und häufig völlig unterschätzte Quelle für Systeminformationen. Die klassischen Ereignislogbücher werden dabei von den Cmdlets aus der Familie *EventLog* verwaltet:

```
PS > Get-Command -noun EventLog
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-EventLog	Clear-EventLog [-LogName] <String[]> [[-...
Cmdlet	Get-EventLog	Get-EventLog [-LogName] <String> [[-Insta...
Cmdlet	Limit-EventLog	Limit-EventLog [-LogName] <String[]> [-Co...
Cmdlet	New-EventLog	New-EventLog [-LogName] <String> [-Source...
Cmdlet	Remove-EventLog	Remove-EventLog [-LogName] <String[]> [[-...
Cmdlet	Show-EventLog	Show-EventLog [[-ComputerName] <String>] ...
Cmdlet	Write-EventLog	Write-EventLog [-LogName] <String> [-Sour...

Mit diesen Cmdlets können die Ereignislogbücher umfassend konfiguriert, ihre Inhalte analysiert und neue Ereignisse darin eingetragen werden.

Sehr viel tiefgreifendere Systeminformationen liefern die dienst- und anwendungsabhängigen Logbücher, die in Windows Vista/Server 2008 eingeführt wurden. Diese können nur über das Cmdlet *Get-WinEvent* abgerufen werden. Über dessen Parameter sind jedoch erstaunlich umfassende Filterungen möglich, wie die Beispiele in diesem Kapitel gezeigt haben. Dazu übergeben Sie dem Cmdlet ein Hashtable, das die Kriterien enthält, nach denen Sie die Ergebnisse filtern möchten.

HINWEIS

Enthalten Logbücher sicherheitsrelevante oder persönlichkeitsrechtliche Informationen, ist ihr Inhalt geschützt und kann nur von Administratoren gelesen werden. Bei den klassischen Logbüchern gilt dies für das *Security*-Logbuch. Bei den dienst- und anwendungsspezifischen Logbüchern zählen beispielsweise Performance- und Diagnosedaten oder auch Informationen zu Gruppenrichtlinien dazu.

Kapitel 12

Zugriffsberechtigungen

In diesem Kapitel:

Zugriffsberechtigungen lesen	364
Zugriffsrechte für Dateien und Ordner festlegen	367
Zugriffsrechte eines Registrierungsschlüssels festlegen	374
Besitz übernehmen	377
Zusammenfassung	379

Windows kontrolliert den Zugriff auf Dateisystem- und Registrierungsobjekte mit Hilfe elektronischer Schlösser. Diese Schlösser werden *Sicherheitsbeschreibung* (engl. Security Descriptor) genannt und sind komplexe, umfangreiche elektronische Schließsysteme, die genau festlegen, welche Personen (oder Gruppen) welchen Zugriff auf die Objekte haben.

PowerShell kann die Einstellungen dieses Schließsystems mit *Get-Acl* lesen und mit *Set-Acl* auch verändern. Dennoch sollten Sie insbesondere Änderungen an diesem Schließsystem nur mit äußerster Vorsicht vornehmen, damit Sie sich nicht versehentlich selbst ausschließen.

Zugriffsberechtigungen lesen

Sie möchten herausfinden, welche Personen Zugriffsrechte auf eine Datei, einen Ordner oder einen Registrierungsschlüssel haben. Sie möchten zum Beispiel prüfen, welche Personen auf Ihr persönliches Benutzerprofil und damit auf Ihre persönlichen Daten zugreifen dürfen.

Lösung

Das Cmdlet *Get-Acl* kann die Sicherheitsbeschreibung sichtbar machen. Die einzelnen Zugriffssteuerungseinträge (Access Control Entries, ACE) finden Sie in der Eigenschaft *Access*. Die folgende Zeile liest die Sicherheitsbeschreibung des Windows-Ordners:

```
PS > Get-Item $env:windir | Get-Acl | Select-Object -expandProperty Access

FileSystemRights : 268435456
AccessControlType : Allow
IdentityReference : ERSTELLER-BESITZER
IsInherited       : False
InheritanceFlags  : ContainerInherit, ObjectInherit
PropagationFlags  : InheritOnly
(...)
```

Wollen Sie die Sicherheitsbeschreibung eines Registrierungsschlüssels anzeigen, ändern Sie lediglich den Pfadnamen:

```
PS > Get-Item HKLM:\Software | Get-Acl | Select-Object -expandProperty Access

RegistryRights    : 268435456
AccessControlType : Allow
IdentityReference : ERSTELLER-BESITZER
IsInherited       : False
InheritanceFlags  : ContainerInherit
PropagationFlags  : InheritOnly
(...)
```

Häufig kann es einfacher (und schneller) sein, für die Verwaltung von Rechten auf externe Anwendungen wie *cacls.exe* oder *icacls.exe* zurückzugreifen. Die meisten dieser Werkzeuge kön-

nen allerdings nur Berechtigungen im Dateisystem lesen, nicht aber in der Registrierungsdatenbank:

```
PS > icacls $env:windir
C:\Windows NT SERVICE\TrustedInstaller:(F)
          NT SERVICE\TrustedInstaller:(CI)(IO)(F)
          NT-AUTORITÄT\SYSTEM:(M)
          NT-AUTORITÄT\SYSTEM:(OI)(CI)(IO)(F)
          VORDEFINIERT\Administratoren:(M)
          VORDEFINIERT\Administratoren:(OI)(CI)(IO)(F)
          VORDEFINIERT\Benutzer:(RX)
          VORDEFINIERT\Benutzer:(OI)(CI)(IO)(GR,GE)
          ERSTELLER-BESITZER:(OI)(CI)(IO)(F)
```

1 Dateien erfolgreich verarbeitet, bei 0 Dateien ist ein Verarbeitungsfehler aufgetreten.

Hintergrund

Zugriffsberechtigungen regeln den Zugriff auf Informationen und legen fest, welche Personen auf welche Weise auf die Informationen zugreifen dürfen. Damit bilden sie einen wichtigen Teil des Sicherheitsfundaments von Windows. Zugriffsberechtigungen werden über Sicherheitsbeschreibungen kontrolliert. *Get-Acl* liest die Sicherheitsbeschreibung eines zugriffsgeschützten Objekts. Dabei kann es sich um eine Datei, einen Ordner oder auch einen Registrierungsschlüssel handeln.

Die Eigenschaft *Owner* liefert den Besitzer des Schlüssels. Der Besitzer kann jederzeit die Zugriffseinstellungen ändern. Andere Benutzer dürfen dies nur, wenn sie über die dafür nötigen Zugriffsrechte verfügen. Die eigentlichen Zugriffsberechtigungen und -verbote finden sich als Liste in der Eigenschaft *Access*. Jeder Eintrag steht dabei stellvertretend für eine Person oder eine Gruppe (festgelegt in der Eigenschaft *IdentityReference*) und bestimmt, welche Art des Zugriffs erlaubt oder verboten ist (festgelegt in den Eigenschaften *FileSystemRights* und *AccessControlType*).

Einige Zugriffsrechte werden im Klartext genannt werden, während andere als Zahl (Bitmaske) aufgeführt werden. PowerShell kann nur die Rechte in Klartext verwandeln, die in der Aufzählung *FileSystemRights* bekannt sind:

```
PS > [System.Enum]::GetNames([System.Security.AccessControl.FileSystemRights])
ListDirectory
ReadData
WriteData
CreateFiles
CreateDirectories
AppendData
ReadExtendedAttributes
WriteExtendedAttributes
(...)
```

Den Zusammenhang erkennt man, wenn man die zugrunde liegende Bitmaske sichtbar macht. Die folgende Zeile liefert die Bitmaske für alle Berechtigungseinträge des Windows-Ordners:

```
PS > Get-Acl $env:windir | Select-Object -ExpandProperty Access | →
ForEach-Object { '{0,70} : {1,-30}' -f [System.Convert]::ToString →
([Int64]$.FileSystemRights,2), $.IdentityReference }
10000000000000000000000000000000 : ERSTELLER-BESITZER
10000000000000000000000000000000 : NT-AUTORITÄT\SYSTEM
100110000000110111111 : NT-AUTORITÄT\SYSTEM
10000000000000000000000000000000 : VORDEFINIERT\Administratoren
100110000000110111111 : VORDEFINIERT\Administratoren
10010000000010101001 : VORDEFINIERT\Benutzer
10000000000000000000000000000000 : NT SERVICE\TrustedInstaller
111110000000111111111 : NT SERVICE\TrustedInstaller
```

Jedem Recht steht also ein bestimmtes Bit gegenüber. Klartextnamen wie *FullControl* fassen häufig benötigte Rechtekombinationen unter einem Namen zusammen. Erscheinen die Zugriffsrechte als Zahl und nicht als Klartextnamen, handelt es sich um eine individuelle Rechtekombination, für die es keinen vordefinierten Begriff gibt.

Möchten Sie wissen, welche Rechte die Klartextnamen zusammenfassen, lassen Sie sich deren Bitmuster anzeigen:

```
PS > [System.Enum]::GetNames([System.Security.AccessControl.FileSystemRights]) | →
ForEach-Object { $value = [System.Enum] →
::Parse([System.Security.AccessControl.FileSystemRights], $ ); →
'{1,70} : {0,-30}' -f $_, [System.Convert]::ToString([Int64]$value,2) }
1 : ListDirectory
1 : ReadData
10 : WriteData
10 : CreateFiles
100 : CreateDirectories
100 : AppendData
1000 : ReadExtendedAttributes
10000 : WriteExtendedAttributes
100000 : Traverse
100000 : ExecuteFile
1000000 : DeleteSubdirectoriesAndFiles
10000000 : ReadAttributes
10000000 : WriteAttributes
100010110 : Write
10000000000000000000 : Delete
10000000000000000000 : ReadPermissions
100000000010001001 : Read
100000000010101001 : ReadAndExecute
110000000110111111 : Modify
10000000000000000000 : ChangePermissions
10000000000000000000 : TakeOwnership
10000000000000000000 : Synchronize
11111000000011111111 : FullControl
```

Die Art der Berechtigungseinträge unterscheidet sich abhängig davon, ob Sie die Sicherheitsbeschreibung eines Dateisystemobjekts (Datei, Ordner) oder eines Registrierungsschlüssels untersuchen. Im Dateisystem wird die Art des geschützten Zugriffs in der Eigenschaft *FileSystemRights* vermerkt. Bei Berechtigungseinträgen für die Registrierungsdatenbank finden sich die Art des Zugriffs dagegen in der Eigenschaft *RegistryRights*. Die übrigen Eigenschaften sind in beiden Fällen gleich.

Eigenschaft	Beschreibung
<i>FileSystemRights</i>	Dateisystem: Beschreibung des Zugriffs
<i>RegistryRights</i>	Registrierungsdatenbank: Beschreibung des Zugriffs
<i>AccessControlType</i>	<i>Allow</i> (dieser Zugriff ist gestattet) oder <i>Deny</i> (Zugriff ist ausdrücklich verboten). Verbote haben Vorrang vor Erlaubnis.
<i>IdentityReference</i>	Person oder Gruppe, für die dieser Eintrag gilt
<i>IsInherited</i>	<i>\$true</i> , wenn diese Einstellung in einem übergeordneten Schlüssel festgelegt und weitervererbt wurde
<i>InheritanceFlags</i>	Legt fest, wie diese Einstellungen an untergeordnete Schlüssel und Werte weitergegeben werden
<i>PropagationFlags</i>	Legt fest, ob diese Einstellungen an untergeordnete Schlüssel und Werte weitergegeben werden können und ob sie auch für das aktuelle Objekt gelten oder nur für die untergeordneten

Tabelle 12.1 Eigenschaften einer Zugriffssteuerungsliste (Access Control List)

Alternativ kann die gesamte Sicherheitsbeschreibung auch in der SDDL (Security Descriptor Definition Language) angezeigt werden. Das Ergebnis ist ein (langer) Textstring, der alle Eigenschaften und Einstellungen der Sicherheitsbeschreibung in codierter Form enthält und sich besonders dafür eignet, Sicherheitsbeschreibungen als Textdatei zu sichern oder von dort wiederherzustellen.

```
PS > (Get-Acl $env:windir).GetSecurityDescriptorSddlForm('A11')
O:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464G:S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464D:PAI(A;OICIIO;GA;;;CO)(A;OICIIO;GA;;;SY)(A;0x1301bf;;;SY)(A;OICIIO;GA;;;BA)(A;0x1301bf;;;BA)(A;OICIIO;GXGR;;;BU)(A;0x1200a9;;;BU)(A;CIO;GA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)(A;FA;;;S-1-5-80-956008885-3418522649-1831038044-1853292631-2271478464)
```

Zugriffsrechte für Dateien und Ordner festlegen

Sie möchten den Zugriff auf Dateien und Ordner festlegen, um beispielsweise sicherzustellen, dass nur bestimmte Personen darauf Zugriff haben.

Lösung

Erstellen Sie eine neue Sicherheitsbeschreibung und weisen Sie sie mittels *Set-Acl* der Datei oder dem Ordner zu, den Sie schützen wollen:

```
PS > md $home\Testordner | Out-Null
PS > $rule1 = New-Object System.Security.AccessControl →
    .FileSystemAccessRule("Administratoren", "FullControl", →
        "ObjectInherit,ContainerInherit", "None", "Allow")
PS > $rule2 = New-Object System.Security.AccessControl. →
    FileSystemAccessRule("Jeder", "Read", →
        "ObjectInherit,ContainerInherit", "None", "Allow")
PS > $acl = Get-Acl $home\Testordner
PS > $acl.SetAccessRule($rule1)
PS > $acl.SetAccessRule($rule2)
PS > $acl.SetAccessRuleProtection($true, $false)
PS > Set-Acl Testordner $acl
```

WICHTIG

Sie benötigen Administratorrechte, um die Sicherheitsbeschreibung zu ändern.

Alternativ verwenden Sie die SDDL-Schreibweise, um die Sicherheitsbeschreibung mit Hilfe von *SetSecurityDescriptorSddlForm()* zu erstellen:

```
PS > $sd = New-Object System.Security.AccessControl.FileSecurity
PS > $sddl = " 0:BAG:BAD:PAI(A;OICI;FR;;;WD)(A;OICI;FA;;;BA)"
PS > $sd.SetSecurityDescriptorSddlForm($sddl)
PS > Set-Acl $home\Testordner $sd
```

Einfacher gelingt die Aufgabe mit Befehlszeilentools wie *cacls* beziehungsweise dessen Nachfolger *icacls*. Diese Programme benötigt im Gegensatz zu den Skriptmethoden nur dann Administratorrechte, wenn die Zugriffsrechte eines Objekts dies tatsächlich erfordern. Möchten Sie als Nicht-Administrator die Sicherheitsbeschreibung eines selbst angelegten Ordners ändern, gelingt dies also nicht mit den Skriptmethoden, unter Umständen wohl aber mit Werkzeugen wie *icacls.exe*.

Die folgende Zeile weist dem Ordner *Testordner* in Ihrem Benutzerprofil *Administratoren* vollen Zugriff und allen anderen Benutzern Lese- und Ausführungsberechtigung zu:

```
PS > icacls $home\Testordner /grant:r Administratoren:F Jeder:RX
Bearbeitete Datei: C:\Users\Tobias\Testordner
1 Dateien erfolgreich verarbeitet, bei 0 Dateien ist ein Verarbeitungsfehler aufgetreten.
```

Hintergrund

Berechtigungen können von übergeordneten Elementen weitervererbt werden, sodass das elektronische Schloss einer Datei oder eines Ordners außerordentlich komplex sein kann. Sie sollten

sich sehr genau überlegen, ob Sie überhaupt mit einer Automationssprache Zugriffsberechtigungen setzen wollen oder dies lieber bei Bedarf von Hand tun.

Häufig ist das Setzen von Zugriffsberechtigungen mit Werkzeugen wie *cacls.exe* bzw. seinem Nachfolger *icacls.exe* einfacher (und sicherer, weil sich hier keine Skriptfehler einschleichen können). Informationen über die Bedienung erhalten Sie mit dem Parameter */?*:

```
PS > cacls /?
PS > icacls /?
```

Das Ergebnis Ihrer Änderungen können Sie im Windows-Explorer überprüfen, indem Sie die Datei oder den Ordner mit der rechten Maustaste anklicken und im Kontextmenü *Eigenschaften* wählen. Holen Sie anschließend im Dialogfeld die Registerkarte *Sicherheit* in den Vordergrund. Dort können Sie die Zugriffsberechtigungen auch manuell ändern (dieses Dialogfeld steht nur bei den Windows Home-Editionen nicht zur Verfügung, wo Sie die Zugriffsberechtigungen allerdings trotzdem per Skript oder Befehlszeilenanweisung setzen und ändern könnten).

Die SDDL-Textrepräsentation der Zugriffsberechtigung besteht aus einer Serie von codierten Kennziffern und Kennzahlen:

```
PS > $sddl = "O:BAG:BAD:PAI(A;CI;KA;;;WD)(A;CI;KA;;;BA)"
```

Die SDDL ist in mehrere Gruppen aufgeteilt:

- **Hinter »O:«** folgt die eindeutige SID oder die Kennung des Besitzers:

```
O:S-1-5-21-2613171836-1965730769-3820153312-1005
O:BA
```

- **Hinter »G:«** folgt die eindeutige SID oder die Kennung der primären Gruppe:

```
G:S-1-5-21-2613171836-1965730769-3820153312-513
G:BA
```

- **Hinter »D:«** folgen zunächst die DACL-Flags, die die Vererbung von übergeordneten Schlüsseln festlegen:

```
D:PAI
```

DACL-Flag	Beschreibung
P	<i>Protected</i> . Es werden keine Berechtigungen von übergeordneten Schlüsseln empfangen
AR	Einstellungen werden automatisch an alle untergeordneten Objekte vererbt

Tabelle 12.2 DACL-Flags (DACL steht für Discretionary Access Control List, besitzerverwaltete Zugriffssteuerungsliste)

DACL-Flag	Beschreibung
AI	Einstellungen können automatisch an alle untergeordneten Objekte vererbt werden, wenn dies festgelegt wird

Tabelle 12.2 DACL-Flags (DACL steht für Discretionary Access Control List, besitzerverwaltete Zugriffssteuerungsliste) (*Fortsetzung*)

Danach folgen die eigentlichen Zugriffssteuerungseinträge (Access Control Entries) in runden Klammern. Sie enthalten durch Semikola getrennt jeweils sechs Angaben. Die erste Information legt fest, ob es sich um eine Erlaubnis (»A«) oder ein Verbot (»D«) handelt:

(A;CI;KA;;;WD)

Kürzel	Beschreibung
A	Erlaubt den Zugriff
D	Verbietet den Zugriff
OA	Erlaubt den Objektzugriff auf ein bestimmtes Objekt, dessen GUID als vierte Information angegeben wird
OD	Verbietet den Objektzugriff auf ein bestimmtes Objekt, dessen GUID als vierte Information angegeben wird
AU	Überwachung
AL	Alarm
OU	Systemüberwachung
OL	Systemalarm

Tabelle 12.3 Access Control Entry-Typ

Die zweite Information legt die Vererbung fest:

(A;**CI**;KA;;;WD)

Kürzel	Beschreibung
CI	Ist das untergeordnete Objekt ein Container (Ordner oder Schlüssel), wird die Berechtigung weitervererbt
OI	Ist das untergeordnete Objekt kein Container (Dateien oder Werte), wird die Berechtigung weitervererbt
NP	Nur die direkten Unterobjekte erhalten die Berechtigungen, aber nicht deren Unterobjekte
IO	Berechtigungen wirken sich nicht auf das aktuelle Objekt aus, sondern werden nur weitervererbt
ID	Berechtigungen wurden von einem übergeordneten Objekt ererbt

Tabelle 12.4 Vererbung festlegen

Kürzel	Beschreibung
SA	Erfolgreicher Zugriff wird protokolliert
FA	Fehlgeschlagener Zugriff wird protokolliert

Tabelle 12.4 Vererbung festlegen (*Fortsetzung*)

Die dritte Information legt die Zugriffsart fest, die hier kontrolliert wird. Erlaubt ist entweder ein Zahlenwert als Bitmaske oder die Kennung eines Standardzugriffs.

(A;CI;KA;;;WD)

Kürzel	Beschreibung
GA	Allgemeiner Vollzugriff
GR	Allgemeiner Lesezugriff
GW	Allgemeiner Schreibzugriff
GX	Allgemeiner Ausführungszugriff

Tabelle 12.5 Standardzugriffsberechtigungen

Kürzel	Beschreibung
RC	Lesen
SD	Löschen
WD	Verändern
WO	Besitzer ändern
RP	Eigenschaften lesen
WP	Eigenschaften schreiben
CC	Untergeordnete Objekte anlegen
DC	Untergeordnete Objekte löschen
LC	Inhalt auflisten
SW	Alle validierten Schreibzugriffe
LO	Objekte auflisten
DT	Rekursiv löschen
CR	Alle erweiterten Rechte
FA	Vollständiger Dateizugriff
FR	Dateien lesen

Tabelle 12.6 Zugriffsberechtigungen im Dateisystem

Kürzel	Beschreibung
FW	Dateien schreiben
FX	Dateien ausführen

Tabelle 12.6 Zugriffsberechtigungen im Dateisystem (*Fortsetzung*)

Kürzel	Beschreibung
KA	Vollzugriff
KR	Lesen
KW	Schreiben
KX	Ausführen

Tabelle 12.7 Zugriffsberechtigungen in der Registrierungsdatenbank

Die vierte und fünfte Information wird nur benötigt, wenn es sich um einen Zugriffssteuerungseintrag vom Typ *OA* oder *OD* handelt. In diesem Fall werden hier die GUID des Objekts genannt, für das dieser Zugriffssteuerungseintrag gelten soll, sowie die GUID des Objekts für die Vererbung.

Die sechste Information schließlich legt die Person oder Gruppe fest, für die dieser Eintrag gelten soll. Erlaubt ist entweder eine SID oder die Kennziffer einer der vordefinierten Gruppen und Personen.

(A;CI;KA;;;WD)

Weil im SDDL die Personen und Gruppen eventuell mit ihren eindeutigen SIDs (SicherheitsIDs) genannt werden, sind SDDLs nicht ohne Weiteres über Domänengrenzen hinweg übertragbar. Sie können allerdings nicht übertragbare SIDs durch allgemein gültige Benutzerkontenkürzel aus Tabelle 12.8 ersetzen und die SDDL so übertragbar machen:

Kürzel	Beschreibung
AO	Kontenoperatoren
AN	Anonyme Anmeldung
AU	Authentifizierte Anwender
BA	Administratoren
BG	Gäste
BO	Sicherungsoperatoren
BU	Benutzer
CA	Zertifikatserver-Administratoren

Tabelle 12.8 Vordefinierte Personen und Gruppen

Kürzel	Beschreibung
CG	Besitzer-Gruppe
CO	Besitzer
DA	Domänen-Admins
DC	Domänencomputer
DD	Domänencontroller
DG	Domänen-Gäste
DU	Domänen-Benutzer
EA	Enterprise-Administratoren
ED	Enterprise-Domänencontroller
WD	Jeder
PA	Gruppenrichtlinien-Administratoren
IU	Interaktiv angemeldete Benutzer
LA	Lokaler Administrator
LG	Lokaler Gast
LS	Lokaler Dienst
SY	Lokales System
NU	Netzwerk-Logon-Benutzer
NO	Netzwerkkonfigurations-Operator
NS	Netzwerk-Dienst
PO	Drucker-Operatoren
PS	Personal Self
PU	Hauptbenutzer
RS	RAS Server Gruppe
RD	Remotedesktopdienste-Benutzer
RE	Replikator
SA	Schema-Administratoren
SO	Server-Operatoren
SU	Service Logon User

Tabelle 12.8 Vordefinierte Personen und Gruppen (*Fortsetzung*)

Zugriffsrechte eines Registrierungsschlüssels festlegen

Sie wollen festlegen, welche Personen oder Gruppen in welcher Art auf einen Registrierungsschlüssel zugreifen dürfen.

Lösung

Legen Sie für den Schlüssel eine eigene Sicherheitsbeschreibung an und weisen Sie diesen mit *Set-Acl* dem Schlüssel zu:

```
PS > md HKCU:\Software\Testschlüssel | Out-Null
PS > $rule1 = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Administratoren", "FullControl", "ObjectInherit,ContainerInherit", →
    "None", "Allow")
PS > $rule2 = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Jeder", "ReadKey", "ObjectInherit,ContainerInherit", "None", "Allow")
PS > $acl = Get-Acl HKCU:\Software\Testschlüssel
PS > $acl.SetAccessRule($rule1)
PS > $acl.SetAccessRule($rule2)
PS > $acl.SetAccessRuleProtection($true, $false)
PS > Set-Acl HKCU:\Software\Testschlüssel $acl
```

Das Ergebnis lässt sich mit dem Registrierungs-Editor überprüfen. Öffnen Sie den Registrierungs-Editor:

```
PS > regedit
```

Navigieren Sie anschließend zum Schlüssel *HKEY_CURRENT_USER\Software\Testschlüssel* und klicken Sie diesen mit der rechten Maustaste an. Wählen Sie dann im Kontextmenü den Eintrag *Berechtigungen*. Ein Dialogfeld öffnet sich und zeigt die Berechtigungen an. Über die Schaltfläche *Erweitert* öffnen Sie eine Ansicht mit den einzelnen Berechtigungseinträgen.

TIPP

Das Setzen von Sicherheitsbeschreibungen in der Registrierungsdatenbank wird von den herkömmlichen Befehlszeilenwerkzeugen *cacls* und *icacls* nicht unterstützt.

Hintergrund

Um den Zugriff auf Registrierungsschlüssel zu regeln, ändern Sie dessen Sicherheitsbeschreibung. Dazu stehen Ihnen zwei verschiedene Wege offen:

- **Neue Sicherheitsbeschreibung** Legen Sie eine neue Sicherheitsbeschreibung an und weisen Sie ihn dem Schlüssel zu
- **SDDL** Legen Sie die Sicherheitseinstellungen in der sogenannten SDDL-Syntax (Security Descriptor Definition Language) fest, erstellen Sie dann daraus eine echte Sicherheitsbeschreibung und weisen Sie diesen dem Schlüssel zu

Im ersten Fall erstellen Sie die Sicherheitsbeschreibung komplett neu. Jeder einzelne Zugriff wird dabei über eine Regel festgelegt, die folgendermaßen aussieht:

```
PS > $person = [System.Security.Principal.NTAccount]"Administratoren"
PS > $zugang = [System.Security.AccessControl.RegistryRights]"FullControl"
PS > $vererbung = [System.Security.AccessControl.InheritanceFlags] →
    "ObjectInherit,ContainerInherit"
PS > $weitergabe = [System.Security.AccessControl.PropagationFlags]"None"
PS > $typ = [System.Security.AccessControl.AccessControlType]"Allow"
PS > $rule = New-Object System.Security.AccessControl.RegistryAccessRule →
    ($person,$zugang,$vererbung,$weitergabe,$typ)
```

Vereinfacht kann man solch eine Regel auch in einer einzigen Zeile erstellen:

```
PS > $rule = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Administratoren", "FullControl", "ObjectInherit,ContainerInherit", →
    "None", "Allow")
```

Wollen Sie die Vererbung und Weitergabe nicht explizit festlegen, geht es noch kürzer:

```
PS > $rule = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Administratoren", "FullControl", "Allow")
```

Neue Regeln können in einen vorhandenen *SecurityDescriptor* mit *SetAccessRule()* und *ResetAccessRule()* eingefügt werden. *ResetAccessRule()* fügt die neue Regel nicht nur ein, sondern entfernt gleichzeitig ältere schon vorhandene Regeln für die Person oder Gruppe, die Sie in Ihrer neuen Regel festgelegt haben.

```
PS > $acl = Get-Acl HKCU:\Software\Testschlüssel
PS > $rule1 = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Administratoren","FullControl","Allow")
PS > $rule2 = New-Object System.Security.AccessControl.RegistryAccessRule →
    ("Jeder","ReadKey","Allow")
PS > $acl.ResetAccessRule($rule1)
PS > $acl.ResetAccessRule($rule2)
PS > $acl
```

Path	Owner	Access
Microsoft.PowerShell.Core...	LT1\Tobias	Jeder Allow ReadKey...

```
PS > $acl.Access

RegistryRights      : ReadKey
AccessControlType   : Allow
IdentityReference   : Jeder
IsInherited         : False
InheritanceFlags    : None
PropagationFlags     : None
```

```
RegistryRights      : FullControl
AccessControlType   : Allow
IdentityReference    : VORDEFINIERT\Administratoren
IsInherited          : False
InheritanceFlags     : None
PropagationFlags     : None
```

Wenn Sie der geänderten Sicherheitsbeschreibung mit *Set-Acl* zurückschreiben, ersetzt er die alten Regeln durch die neuen. Allerdings bleibt die Vererbung erhalten und Ihr Schlüssel empfängt nach wie vor die Berechtigungen der übergeordneten Schlüssel.

Um diese Vererbung abzuschalten, müssen Sie mit *SetAccessRuleProtection()* zuerst die Vererbung Ihren Wünschen entsprechend neu konfigurieren:

```
PS > $acl.SetAccessRuleProtection($true, $false)
PS > Set-Acl HKCU:\Software\Testschlüssel $acl
```

ACHTUNG Wenn Sie den Testschlüssel zuvor bereits gesichert haben, verfügen Sie möglicherweise nicht mehr über die Berechtigung, die Sicherheitseinstellungen zu ändern. Erhalten Sie also eine Fehlermeldung wegen mangelnder Zugriffsrechte, stellen Sie sicher, dass Sie Ihr PowerShell-Skript mit vollen Administratorrechten ausführen und über die notwendigen Zugriffsrechte verfügen.

Welche Einstellmöglichkeiten Ihnen bei der Erstellung neuer Sicherheitsregeln zur Verfügung stehen, finden Sie heraus, indem Sie sich die erlaubten Werte ausgeben lassen. Die Berechtigungen werden zum Beispiel mit dem Typ *[System.Security.AccessControl.RegistryRights]* angegeben:

```
PS > [System.Enum]::GetNames([System.Security.AccessControl.RegistryRights])
QueryValues
SetValue
CreateSubKey
EnumerateSubKeys
Notify
CreateLink
Delete
ReadPermissions
WriteKey
ExecuteKey
ReadKey
ChangePermissions
TakeOwnership
FullControl
```

Die möglichen Vererbungseinstellungen liefert diese Zeile:

```
PS > [System.Enum]::GetNames([System.Security.AccessControl.InheritanceFlags])
None
ContainerInherit
ObjectInherit
```


Und die Typen der Zugriffssteuerungseinträge (Access Control Entry) verrät diese Zeile:

```
PS > [System.Enum]::GetNames([System.Security.AccessControl.AccessControlType])
Allow
Deny
```

Möchten Sie mehrere Einstellmöglichkeiten miteinander kombinieren, geben Sie sie komma-separiert an:

```
PS > $vererbung = [System.Security.AccessControl.InheritanceFlags] -->
"ObjectInherit,ContainerInherit"
```

In der Praxis ist es häufig sehr viel leichter, Sicherheitsbeschreibungen in der sogenannten SDDL-Form festzulegen. Die Einstellungen werden dabei in Textform angegeben. Sie können die SDDL-Form zum Beispiel aus einem vorhandenen Schlüssel auslesen, dessen Berechtigungen Sie vorher genau so eingestellt haben, wie Sie es erfordern.

Legen Sie sich zum Beispiel im Schlüssel *HKEY_CURRENT_USER\Software* einen neuen Schlüssel namens *Prototyp* an, und weisen Sie diesem Schlüssel dann über die grafische Oberfläche des Registrierungs-Editors die benötigten Zugriffsrechte zu. Danach rufen Sie die Einstellungen in SDDL-Form ab:

```
PS > $sd = Get-Acl HKCU:\Software\Prototyp
PS > $sd.sddl
O:S-1-5-21-2613171836-1965730769-3820153312-1005G:S-1-5-21-2613171836-1965730769-3820153312-513D:PAI(A;CI;KA;;;WD)(A;CI;KA;;;BA)
```

Diese Textrepräsentation eines *SecurityDescriptors* kann mit *SetSecurityDescriptorSddlForm()* in eine Sicherheitsbeschreibung eingetragen und danach Objekten mit *Set-Acl* zugewiesen werden. Im folgenden Beispiel wird also die SDDL des Prototypschlüssels auf einen anderen übertragen. Wenn Sie dieses Beispiel nachvollziehen möchten, ändern Sie die SDDL entsprechend.

```
PS > $sd = New-Object System.Security.AccessControl.RegistrySecurity
PS > $sddl = "O:S-1-5-21-2613171836-1965730769-3820153312-1005G:S-1-5-21-2613171836-1965730769-3820153312-513D:PAI(A;CI;KA;;;WD)(A;CI;KA;;;BA)"
PS > $sd.SetSecurityDescriptorSddlForm($sddl)
PS > Set-Acl HKCU:\Software\Testschlüssel $sd
```

ACHTUNG

Sie benötigen Administratorrechte, um Sicherheitsberechtigungen zu ändern.

Besitz übernehmen

Sie möchten den Besitz an einem Ordner übernehmen, zum Beispiel, weil Sie keine Zugriffsberechtigungen (mehr) auf diesen Ordner haben und die Zugriffsberechtigungen wiederherstellen wollen.

Lösung

Verwenden Sie die WMI-Klasse *Win32_Directory*, um auf den Ordner zuzugreifen, und nutzen Sie die *TakeOwnership()*-Methode dieser Klasse, um den Besitz zu übernehmen. Die folgende Funktion *Set-Ownership* erwartet einen gültigen Pfadnamen zu einem Ordner und versucht dann, den Besitz zu übernehmen. Die Funktion liefert als Rückgabewert 0, falls der Besitz übernommen werden konnte, und andernfalls einen WMI-Fehlercode.

```
Function Set-Ownership($pfad) {
    If ( (Test-Path $pfad) -eq $false) →
        { Throw "Ordner $pfad nicht gefunden." }
    $pfad = $pfad.Replace('\', '\\')
    $wql = "Select * from Win32_Directory Where Name = '$pfad'"
    Get-WmiObject -query $wql | ForEach-Object { →
        ($_.TakeOwnership()).ReturnValue }
}
```

```
PS > Set-Ownership $home\testordner
```

Ab Windows Vista/Server 2008 steht Ihnen alternativ das Befehlszeilentool *takeown.exe* zur Verfügung:

```
PS > takeown /F $home\testordner
```

```
ERFOLGREICH: Die Datei (oder der Ordner) "C:\Users\w7-pc9\testordner" gehört jetzt dem Benutzer "DEM05\w7-pc9".
```

Mit den Optionen dieses Werkzeug kann unter anderem auf Wunsch auch das Konto der Person angegeben werden, die den Besitz übernehmen soll. Eine Übersicht aller Optionen erhalten Sie so:

```
PS > takeown /?
```

Hintergrund

Der Besitzer einer Datei oder eines Ordners hat das Recht, sämtliche Zugriffsrechte zu ändern. Auf diese Weise verhindert Windows, dass sich Personen ausschließen können.

Gelingt die Besitzübernahme nicht, liefert WMI einen Fehlercode mit den Gründen zurück. Eine Klartextbeschreibung dieser Fehlercodes erhalten Sie zum Beispiel im Internet, indem Sie in einer Internet-Suchmaschine diese Schlüsselwörter eingeben: »Win32_Directory WMI Take-Ownership«.

Alternativ stehen Ihnen neben *takeown.exe*, das erst ab Windows Vista/Server 2008 verfügbar ist, auch die einschlägigen Konsolenbefehle *cacls* und *icacls* zur Verfügung. Die folgende Zeile würde den Besitz an Freds Ordner *Documents* an den Benutzer »Lieselotte« übergeben:

```
PS > icacls c:\users\fred\documents /setowner Lieselotte
```

Zusammenfassung

Windows sichert den Zugang zu Elementen im Dateisystem und der Registrierungsdatenbank mit Sicherheitsbeschreibungen ab, die festlegen, welche Personen welchen Zugang dazu haben sollen. PowerShell kann diese Sicherheitsbeschreibungen mit *Get-Acl* lesen und mit *Set-Acl* neu schreiben. Allerdings müssen Sie hierzu die teils kryptischen Sicherheitsinformationen selbst interpretieren.

Das Setzen neuer Berechtigungen mit *Set-Acl* ist nur eingeschränkt praxistauglich. Erstens kann dieses Cmdlet Sicherheitsbeschreibungen nur komplett ersetzen, aber nicht erweitern. Um die Beschreibungen zu erweitern, müssten Sie sie zuerst mit *Get-Acl* lesen und dann die notwendigen Regeln mit Low-Level-.NET-Methoden einfügen. Anschließend kann die veränderte Sicherheitsbeschreibung mit *Set-Acl* zurückgeschrieben werden. Zweitens erfordert *Set-Acl* immer Administratorrechte, also auch dann, wenn das Objekt, das Sie sichern wollen, Ihnen selbst gehört.

Immerhin ein relevanter Einsatzbereich von *Set-Acl* ist das Schreiben initialer Sicherheitsbeschreibungen für neu angelegte Objekte. Wenn Sie beispielsweise einen neuen Ordner anlegen, können Sie diesem danach mit *Set-Acl* eine Sicherheitsbeschreibung zuweisen. Am einfachsten gelingt dies, wenn die Sicherheitsbeschreibung im sogenannten *SDDL*-Format als Text angegeben wird.

In der Praxis ist es meistens einfacher, zum Lesen und Schreiben von Sicherheitsbeschreibungen die vorhandenen Tools *cacls.exe* oder *icacls.exe* einzusetzen, die nahtlos in PowerShell-Code integriert werden können. Mit ihnen kann man auch neue Berechtigungen in vorhandene Sicherheitsbeschreibungen einfügen oder einzelne Berechtigungen daraus entfernen. Allerdings unterstützen beide Tools nur Objekte im Dateisystem.

Schließlich besteht noch die Möglichkeit, Sicherheitsbeschreibungen direkt mit den in .NET Framework vorhandenen Programmierschnittstellen anzusprechen. In diesem Fall stehen Ihnen sämtliche Möglichkeiten offen, aber weil es sich hierbei um einen Low-Level-Ansatz handelt, ist sehr viel Code und Fachwissen dafür erforderlich.

Kapitel 13

Digitale Zertifikate und Sicherheit

In diesem Kapitel:

Skriptausführung mit <i>ExecutionPolicy</i> erlauben	382
Installiertes Codesigning-Zertifikat auswählen	384
Neues Zertifikat aus Datei installieren	386
Zertifikat aus einer Datei lesen	388
Selbstsigniertes Testzertifikat erstellen	389
Prüfen, ob ein Zertifikat vertrauenswürdig ist	393
Selbstsigniertes Zertifikat für vertrauenswürdig erklären	394
PowerShell-Skript signieren	396
Skriptsignatur überprüfen	399
Zertifikat als <i>.cer</i> -Datei exportieren	401
Zertifikat als <i>.pfx</i> -Datei exportieren	402
Zertifikat löschen	403
Zusammenfassung	405

PowerShell nutzt digitale Signaturen, um PowerShell-Skripts zu sichern. Über die sogenannte *ExecutionPolicy* legen Sie fest, ob Skripts ausgeführt werden dürfen und falls ja, ob Skripts eine gültige digitale Unterschrift besitzen müssen. Was eine solche digitale Unterschrift eigentlich ist, wie Sie Skripts signieren und die dafür notwendigen Zertifikate anlegen, verwalten und löschen, lesen Sie in diesem Kapitel.

Skriptausführung mit *ExecutionPolicy* erlauben

Sie wollen dafür sorgen, dass auf Ihrem Computer (oder in Ihrem Unternehmen) Skripts ausgeführt werden können.

Lösung

Fabrikfrisch erlaubt PowerShell die Ausführung von PowerShell-Skripts nicht. Um die Ausführung von Skripts für den aktuellen Anwender zuzulassen, stellen Sie die *ExecutionPolicy* folgendermaßen ein:

```
PS > Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

Diese Einstellung gilt dauerhaft, bis Sie sie erneut ändern. Lokal gespeicherte Skripts dürfen nun ausgeführt werden. Skripts von potenziell unsicheren Orten (aus dem Internet heruntergeladen, auf Dateiservern außerhalb der eigenen Domäne gespeichert) müssen über eine gültige digitale Signatur verfügen, um ausgeführt zu werden. Möchten Sie die Überprüfung von Signaturen gänzlich ausschalten, verwenden Sie anstelle der Einstellung *RemoteSigned* die Einstellung *Bypass*. Ein Unternehmen kann diese Sicherheitseinstellung auch zentral über Gruppenrichtlinien unternehmensweit festlegen. Sobald Gruppenrichtlinien die *ExecutionPolicy* festlegen, haben manuelle Änderungen keine Auswirkung mehr.

Hintergrund

PowerShell unterscheidet verschiedene *ExecutionPolicies*, die ein Administrator pro Computer festlegt.

```
PS > [System.Enum]::GetNames([Microsoft.PowerShell.ExecutionPolicy])
Unrestricted
RemoteSigned
AllSigned
Restricted
Default
Bypass
Undefined
```

Mit *Get-ExecutionPolicy* erfahren Sie, welche Einstellung aktuell wirksam ist:

```
PS > Get-ExecutionPolicy
RemoteSigned
```

Eine Gesamtübersicht über sämtliche Ebenen der *ExecutionPolicy* erhalten Sie mit dem Parameter *-List*:

```
PS > Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
----	-----
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	RemoteSigned
LocalMachine	Restricted

Die einzelnen Einstellungen der *ExecutionPolicy* in dieser Liste werden nach absteigender Wichtigkeit aufgeführt. Die obersten beiden Scopes entsprechen Gruppenrichtlinieneinstellungen, die ein Unternehmen möglicherweise zentral festgelegt hat. Diese beiden Einstellungen können nur über Gruppenrichtlinien festgelegt werden, nicht aber manuell über *Set-ExecutionPolicy*.

Der Scope *Process* bezieht sich auf die aktuelle PowerShell-Sitzung und ist also nur vorübergehend in dieser Sitzung wirksam. *CurrentUser* legt die Einstellung nur für den aktuellen Benutzer fest und Einstellungen im Scope *LocalMachine* wirken sich für alle Benutzer des jeweiligen Computers aus. Die effektive (tatsächlich wirksame) Einstellung ist die erste Einstellung in der Liste, die nicht auf *Undefined* gesetzt ist.

Die Einstellung der *ExecutionPolicy* bestimmt, wie PowerShell vorgeht, wenn ein PowerShell-Skript gestartet werden soll:

Einstellung	Beschreibung
<i>Unrestricted</i>	Signaturen von PowerShell-Skripts werden grundsätzlich nicht automatisch überprüft. Stammen Skripts von potenziell unsicheren Orten, erscheint jedoch stets eine Sicherheitsabfrage.
<i>Bypass</i>	Die gesamte Überprüfung der Skriptsignaturen wird abgeschaltet. Dies kann sinnvoll sein, wenn Sie Skripts häufiger von fremden Dateiservern starten und die dabei auftretenden Sicherheitsabfragen umgehen wollen. Allerdings werden Sie nun grundsätzlich nicht mehr gewarnt, wenn Skripts von potenziell unsicheren Orten stammen.
<i>RemoteSigned</i>	Skripts, die aus dem Internet oder anderen nicht vertrauten Netzwerken/Dateiservern heruntergeladen wurden, müssen über eine gültige Signatur verfügen, um ausführbar zu sein

Tabelle 13.1 Verschiedene Einstellungen für *ExecutionPolicy*

Einstellung	Beschreibung
<i>AllSigned</i>	Sämtliche PowerShell-Skripts müssen über gültige Signaturen verfügen. Diese Einstellung bietet hohe Sicherheit bei gleichzeitig hohem Aufwand, weil in diesem Fall dafür gesorgt werden muss, dass alle PowerShell-Skripts gültig signiert sind. Man setzt diese Einstellung häufig selektiv für besonders geschäftskritische oder anderweitig sicherheitssensible Server ein.
<i>Restricted</i>	PowerShell-Skripts dürfen nicht ausgeführt werden. Dies ist die Grundeinstellung. Sie ist im Sinne der Sicherheitsphilosophie-Regel der geringsten Rechte (Least Privilege) richtig, um Angriffsflächen zu vermeiden, solange Sie PowerShell (noch) nicht als Automationsprache einsetzen.
<i>Default</i>	Standardeinstellung des Systems, normalerweise <i>Restricted</i>

Tabelle 13.1 Verschiedene Einstellungen für *ExecutionPolicy* (Fortsetzung)

Installiertes Codesigning-Zertifikat auswählen

Sie möchten herausfinden, ob Sie Zugriff auf ein Codesigning-Zertifikat haben, und dieses gegebenenfalls dazu verwenden, ein PowerShell-Skript zu signieren.

Lösung

Alle installierten Zertifikate liegen im Zertifikatspeicher, den Powershell über sein Laufwerk *cert:* verfügbar macht. Listen Sie darin mit *Get-ChildItem* (Alias *dir*, *ls*) alle Codesigning-Zertifikate mit den Parametern *-recurse* und *-codeSigningCert* heraus:

```
PS > Get-ChildItem cert:\ -recurse -codeSigningCert
```

Suchen Sie nur eigene, persönliche Codesigning-Zertifikate, könnten Sie auch direkt in Ihrem eigenen Zertifikatspeicher suchen und benötigen dann keine Rekursion:

```
PS > Get-ChildItem cert:\CurrentUser\My -codeSigningCert
```

Verfügen Sie über mehr als ein Codesigning-Zertifikat, wählen Sie das gewünschte zum Beispiel mit einem Filter (*Where-Object*) aus:

```
PS > $zertifikat = Get-ChildItem cert:\CurrentUser\My -codeSigningCert | →
Where-Object { $_.Subject -eq 'CN=Tobias' }
```

Kennen Sie den Thumbprint des gewünschten Zertifikates, können Sie es auch direkt unter Angabe des Thumbprints oder eines Teiles davon ansprechen. Die folgende Zeile würde das persönliche Zertifikat lesen, dessen Thumbprint mit der Ziffernfolge »19F3A3« beginnt:

```
PS > $zertifikat = Get-ChildItem cert:\CurrentUser\My\19F3A3*
```


Falls diese Befehlszeilen bei Ihnen keine Resultate liefern, verfügen Sie vermutlich noch über kein Codesigning-Zertifikat. Lassen Sie sich entweder von Ihrer Unternehmens-IT ein solches Zertifikat zuteilen oder legen Sie zu Testzwecken ein sogenanntes »selbstsigniertes« Zertifikat an (siehe den Abschnitt »Selbstsigniertes Testzertifikat erstellen« ab Seite 389).

Hintergrund

Zertifikate repräsentieren eine elektronische Identität. Dazu finden sich im Zertifikat eine Reihe öffentlich lesbarer Angaben wie zum Beispiel der Name der Person, der Gültigkeitszeitraum des Zertifikats und sein Verwendungszweck zur Verfügung. Diese können sichtbar gemacht werden, indem Sie das Zertifikat an *Select-Object* weiterleiten:

```
PS > Get-ChildItem cert:\CurrentUser\My | Select-Object *
```

Zertifikate können für unterschiedliche Aufgaben eingesetzt werden, beispielsweise zur Signatur von E-Mails, ausführbarem Code oder auch zur Anmeldung an Netzwerken. Für welche Zwecke ein Zertifikat ausgestellt wurde, ist darin in der Eigenschaft *Extensions* festgelegt. So wird verhindert, dass ein Zertifikat für einen anderen Zweck als denjenigen eingesetzt wird, für das das Zertifikat ursprünglich herausgegeben wurde.

Möchten Sie PowerShell-Skripts mit einer digitalen Signatur versehen, benötigen Sie ein Zertifikat mit dem Verwendungszweck »Codesignatur«. Über den Parameter *-CodeSigningCert* liefert PowerShell nur solche Zertifikate, deren Verwendungszweck »Codesignatur« lautet und bei denen Sie über einen privaten Schlüssel verfügen, um mit Hilfe des Zertifikats Signaturen anzulegen.

HINWEIS

Auch das *Details*-Dialogfeld eines Zertifikats kann von PowerShell aus geöffnet werden. Der folgende Code wählt zunächst das erstbeste Codesigning-Zertifikat aus Ihrem persönlichen Zertifikatspeicher und öffnet dann das Dialogfeld. Die dafür nötige .NET-Klasse wird vorher aus *System.Security* nachgeladen.

```
PS > $zert = Get-ChildItem cert:\CurrentUser\My | Select-Object -first 1
PS > [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > [System.Security.Cryptography.X509Certificates.X509Certificate2UI] →
    ::DisplayCertificate($zert)
```

Möchten Sie Zertifikate von Fall zu Fall aus einem Dialogfeld auswählen, greifen Sie auf .NET Framework zurück:

```

PS > $Store = New-Object system.security.cryptography.X509Certificates.x509Store →
("My", "CurrentUser")
PS > $store.Open("ReadOnly")
PS > [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > $certificate = [System.Security.Cryptography.x509Certificates →
.X509Certificate2UI]::SelectFromCollection($store.certificates, →
"Ihre Zertifikate", "Bitte wählen", 0)
PS > $certificate

```

Thumbprint	Subject
-----	-----
19F3A36BE3584DD49DB375C72FED8B2C9CF10F4D	CN=Tobias

Allerdings zeigt das Dialogfeld alle im angegebenen Zertifikatspeicher vorhandenen Zertifikate an, nicht nur die Zertifikate für das Codesigning. Sie erkennen den Verwendungszweck eines Zertifikats im Dialogfeld in der Spalte *Beabsichtigte Zwecke*:

Neues Zertifikat aus Datei installieren

Sie möchten ein Zertifikat aus einer *.cer*- oder *.pfx*-Datei installieren, zum Beispiel, weil Sie das Zertifikat zuvor auf einem anderen Computer exportiert haben, um es nun auch auf einem zweiten Computer einzusetzen, oder weil Ihnen das Zertifikat als Datei übergeben wurde.

Lösung

Öffnen Sie die *.cer*- oder *.pfx*-Datei. Automatisch erscheint ein Assistent, der Ihnen dabei hilft, das Zertifikat zu installieren.

```

PS > .\zertifikat.cer
PS > .\zertifikat.pfx

```

ACHTUNG Wenn Sie eine *.pfx*-Datei importieren, wird dabei ein Zertifikat mitsamt privatem Schlüssel installiert. Dabei werden Sie aufgefordert, ein Kennwort einzugeben. Gefragt ist dann dasselbe Kennwort, das Sie beim Exportvorgang festgelegt hatten, um den Inhalt der *.pfx*-Datei zu schützen. Der Assistent stellt Ihnen bei *.pfx*-Dateien eine Reihe wichtiger Optionen zur Auswahl. Aktivieren Sie die Option der hohen Sicherheit, muss das Kennwort jedes Mal erneut eingegeben werden, wenn auf den privaten Schlüssel zugegriffen wird, also zum Beispiel jedes Mal, wenn Sie damit ein PowerShell-Skript signieren. Diese Option ist allenfalls dann sinnvoll, wenn Ihr Computer relativ öffentlich ist und Sie keine anderen Schutzmaßnahmen getroffen haben, um Unbefugte von ihm fernzuhalten.

Entscheiden Sie sich außerdem, ob der private Schlüssel exportierbar sein soll oder nicht. Ist er exportierbar, kann der Anwender den privaten Schlüssel wieder seinerseits in eine *.pfx*-Datei verpacken und auf andere Computer übertragen. Ist das unerwünscht, stellen Sie sicher, dass der private Schlüssel als nicht exportierbar markiert wird.

Hintergrund

Zertifikate werden stets in einem Zertifikatspeicher aufbewahrt. Um also ein Zertifikat aus einer Datei zu importieren, benötigen Sie zuerst Zugriff auf den passenden Zertifikatspeicher. Diesen erreichen Sie über ein *X509Certificates.X509Store*-Objekt. Weil sämtliche Objekte rund um Zertifikate aus einer .NET-Assembly stammen, die von PowerShell normalerweise nicht benötigt wird, müssen Sie diese Assembly zuerst mit *LoadWithPartialName()* laden, bevor Sie mit *New-Object* ein neues *X509Certificates.X509Store*-Objekt anlegen können. Geben Sie dabei an, welchen Zertifikatspeicher Sie ansprechen wollen. Für den Zertifikatspeicher Ihrer persönlichen Zertifikate lautet der Speichername *My* (Tabelle 13.2 auf Seite 396) und der Ort *CurrentUser*.

TIPP

Auf gleiche Weise können Sie Zertifikate natürlich auch in andere Speicher importieren. Möchten Sie zum Beispiel ein Stammzertifikat installieren, lautet der Name für die vertrauenswürdigen Stammzertifizierungsstellen *Root*, der Ort *LocalMachine*.

Der Zertifikatspeicher, auf den Sie nun über die Variable *\$store* Zugriff haben, wird als Nächstes mit Lese- und Schreibrechten geöffnet. Dazu benötigen Sie Administratorrechte.

Als Nächstes muss die Zertifikatdatei in ein echtes Zertifikat umgewandelt werden, denn dem Zertifikatspeicher können nur Zertifikate, aber keine Zertifikatdateien direkt hinzugefügt werden.

Für die Umwandlung der Zertifikatdatei in ein Zertifikat legen Sie sich zunächst ein *X509Certificates.X509Certificate2Collection*-Objekt an. Dieses verfügt über die Methode *Import()*, mit der man Zertifikatdateien in Zertifikate verwandeln kann.

Für *.cer*-Dateien braucht *Import()* nur der Pfadname der Zertifikatdatei übergeben zu werden. Bei *.pfx*-Dateien sind drei Angaben wichtig: der Pfadname der *.pfx*-Datei, das Kennwort, mit dem der Inhalt der *.pfx*-Datei geschützt ist, sowie ein sogenanntes *X509KeyStorageFlag*. Es legt fest, ob der private Schlüssel des Zertifikats exportierbar sein soll oder nicht. Wählen Sie *PersistKeySet* für nicht exportierbare und *Exportable* für exportierbare private Schlüssel. Die weiteren möglichen Flags listet die Aufzählung *X509Certificates.X509KeyStorageFlags* auf:

```
PS > [System.Enum]::GetNames([System.Security.Cryptography.X509Certificates
    .X509KeyStorageFlags])
DefaultKeySet
UserKeySet
MachineKeySet
Exportable
UserProtected
PersistKeySet
```

Nach dem Importvorgang enthält das *X509Certificates.X509Certificate2Collection*-Objekt genau ein Zertifikat. Dieses kann dann mit *Add()* dem Zertifikatspeicher hinzugefügt werden, indem Sie den Index des Zertifikats angeben: Weil es das erste (und einzige) in der Collection ist, lautet sein Index 0.

PowerShell kann Zertifikatdateien auch programmgesteuert installieren. Der folgende Code installiert das Zertifikat aus der Datei *zertifikat.cer* ohne privaten Schlüssel in den Speicher Ihrer persönlichen Zertifikate. Diese Datei muss bereits existieren.

```
PS > $filename = "$home\zertifikat.cer"
PS >
PS > [void][System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > $Store = New-Object System.Security.Cryptography.X509Certificates.X509Store →
("My", "CurrentUser")
PS > $store.Open("ReadWrite")
PS > $container = New-Object System.Security.Cryptography.X509Certificates →
.X509Certificate2Collection
PS > $container.Import($filename)
PS > $store.Add($container[0])
PS > $store.Close()
```

Und mit diesem Code würden Sie ein Zertifikat mitsamt privatem Schlüssel aus der Datei *zertifikat.pfx* in den Speicher Ihrer persönlichen Zertifikate installieren. Auch diese Datei muss bereits existieren.

```
PS > $filename = "$home\zertifikat.pfx"
PS >
PS > [void][System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > $Store = New-Object System.Security.Cryptography.X509Certificates →
.x509Store("My", "CurrentUser")
PS > $store.Open("ReadWrite")
PS > $container = New-Object System.Security.Cryptography.X509Certificates. →
X509Certificate2Collection
PS > $container.Import($filename, 'strenggeheim', 'PersistKeySet')
PS > $store.Add($container[0])
PS > $store.Close()
```

Möchten Sie den privaten Schlüssel als exportierbar markieren, ersetzen Sie im Code *PersistKeySet* durch *Exportable*.

Zertifikat aus einer Datei lesen

Sie möchten ein Zertifikat aus einer *.pfx*-Datei lesen, um es anschließend einzusetzen. Sie wollen das Zertifikat aber nicht dauerhaft auf dem Computer installieren.

Lösung

Laden Sie das Zertifikat mit *Get-PfxCertificate* in eine Variable:

```
PS > $filename = "C:\my_cert.pfx"
PS > $cert = Get-PfxCertificate $filename
PS > $cert | Select-Object *
```

Anschließend sind alle Eigenschaften des Zertifikats lesbar, und Sie können es beispielsweise mit *Set-AuthenticodeSignature* dazu einsetzen, Skripts und andere Dateien zu signieren.

Hintergrund

Eine *.pfx*-Datei entspricht einem Zertifikat mit seinem privaten Schlüssel, enthält also alle Informationen, um mit diesem Zertifikat Dateien zu signieren. Da es sich hierbei oft um höchst sensible Informationen handelt, sind die meisten *.pfx*-Dateien mit einem Kennwort geschützt, das eingegeben werden muss, bevor das Zertifikat und sein privater Schlüssel gelesen werden können.

Get-PfxCertificate eignet sich ausgezeichnet dazu, Skripts und andere Dateien mit Zertifikaten zu signieren, die nicht auf dem Computer installiert sind.

Selbstsigniertes Testzertifikat erstellen

Sie möchten sich ein Codesigning-Zertifikat selbst ausstellen.

Lösung

Selbstsignierte Testzertifikate können mit dem (kostenlosen) Werkzeug *makecert.exe* von Microsoft erstellt werden. Laden Sie sich dazu das kostenlose Microsoft *.NET Framework 2.0 Software Development Kit (SDK)* von <http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx> herunter (ca. 400 MB) und installieren Sie es. So erstellen Sie dann mit *makecert.exe* ein selbstsigniertes Zertifikat namens *PowerShellTestCert*:

```
PS > $name = "PowerShellTestCert"
PS > . "$env:programfiles\Microsoft Visual Studio 8\SDK\v2.0\Bin\makecert.exe" ->
-pe -r -n "CN=$name" -eku 1.3.6.1.5.5.7.3.3 -ss 'my' -a sha1
Succeeded
PS > dir cert:\ -recurse -codeSigningCert
```

Verzeichnis: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint	Subject
-----	-----
44F3D50ED03FD846ED11FC5295D100DA736B1D01	CN=PowerShellTestCert
19F3A36BE3584DD49DB375C72FED8B2C9CF10F4D	CN=Tobias

Benötigen Sie von vornherein vertrauenswürdige Zertifikate, legen Sie sich zuerst ein selbstsigniertes Rootzertifikat an. In dessen Namen stellen Sie dann die eigentlichen Zertifikate aus, die nun nicht mehr selbstsigniert sind, sondern von Ihrem Rootzertifikat stammen:

```

PS > $rootPrivateKey = "$home\root.pvk"
PS > $rootCert = "$home\root.cer"
PS > $name = 'AbteilungsRootZertifikat'
PS > . "$env:programfiles\Microsoft Visual Studio 8\SDK\v2.0\Bin\makecert.exe" →
    -n "CN=$name" -a sha1 -eku 1.3.6.1.5.5.7.3.3 -r -sv $rootPrivateKey →
        $rootCert -ss Root -sr LocalMachine
Succeeded
PS > dir $home\root.*

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias

Mode                LastWriteTime         Length Name
----                -
-a---             01.02.2011    10:10           564 root.cer
-a---             01.02.2011    10:10           636 root.pvk

PS > $name = 'Mitarbeiter1'
PS > . "$env:programfiles\Microsoft Visual Studio 8\SDK\v2.0\Bin\makecert.exe" →
    -pe -n "CN=$name" -a sha1 -eku 1.3.6.1.5.5.7.3.3 -ss My -iv $rootPrivateKey →
    -ic $rootCert
Succeeded

```

ACHTUNG Wenn Sie dem Beispiel folgen, erhalten Sie zwei Dateien: *root.pvk* und *root.cer*. Die Datei *root.pvk* kann mit einem Kennwort geschützt werden, das automatisch erfragt wird, wenn diese Datei angelegt wird. Diese beiden Dateien sind ausgesprochen wichtig und sicherheitskritisch. Jeder, der im Besitz der Datei *root.pvk* ist und das zugeordnete Kennwort kennt, kann im Namen dieses Rootzertifikats weitere Nutzerzertifikate erstellen. Deshalb sollten Sie beide Dateien sogleich auf einen Datenträger verschieben und an einem sicheren Ort verwahren.

Hintergrund

Normalerweise beschaffen Sie sich Codesigningzertifikate von einer autorisierten Quelle.

- **Zertifizierungsunternehmen** Das kann ein Zertifizierungsunternehmen wie *VeriSign* oder *Thawte* sein. In diesem Fall überprüft das Zertifizierungsunternehmen gegen Gebühr Ihre Identität und überlässt Ihnen anschließend ein Codesigning-Zertifikat, das für eine gewisse Zeit gültig ist (meist ein bis fünf Jahre). Der Vorteil hierbei: Weil die global agierenden Zertifizierungsunternehmen weltweit anerkannt sind, wird das Codesigning-Zertifikat weltweit überwiegend als vertrauenswürdig eingestuft und akzeptiert.
- **Eigene PKI** Größere Firmen betreiben ihre eigene PKI (Public Key-Infrastruktur). In diesem Fall beantragen Sie das Codesigning-Zertifikat intern bei Ihrer PKI. Vorteil hierbei: Die Firma spart Kosten, wenn viele Zertifikate im Unternehmen eingesetzt werden (zum Beispiel auch für SmartCards, E-Mail-Verschlüsselung etc.). Nachteil: Weil die Zertifikate von der internen Firmen-PKI ausgestellt wurden, gelten sie nur innerhalb der Firma als vertrauenswürdig. In Kundenbeziehungen außerhalb der Firma sind sie meist wenig wert.

Alternativ dazu können Sie sich Zertifikate mit Tools wie *makecert.exe* selbst erstellen. Weil solche Zertifikate aber von jedermann erstellt werden können, lassen sie sich erst einsetzen, wenn Sie sie für vertrauenswürdig erklärt haben. Dazu muss der Herausgeber des Zertifikats im Zerti-

figatspeicher der vertrauenswürdigen Stammzertifizierungsstellen aufgeführt sein, wofür Administratorrechte nötig sind.

Legen Sie ein selbstsigniertes Zertifikat an, ist dieses Zertifikat automatisch auch sein Herausgeber. Selbstsignierte Zertifikate werden vertrauenswürdig, sobald Sie das Zertifikat in den Speicher der vertrauenswürdigen Stammzertifizierungsstellen kopieren.

Alternativ können Sie ein eigenes Rootzertifikat direkt im Speicher der vertrauenswürdigen Stammzertifizierungsstellen anlegen. *makecert.exe* liefert Ihnen dabei eine (optional kennwortgeschützte) *.pvk*-Datei an, in der der geheime private Schlüssel des Rootzertifikats gespeichert ist. Ihn benötigt man, um mit diesem Rootzertifikat weitere Nutzerzertifikate auszustellen. Außerdem erhalten Sie eine *.cer*-Datei mit dem öffentlichen Fingerabdruck (Thumbprint) des Rootzertifikats.

Weil sich Ihr selbstsigniertes Rootzertifikat bereits im Speicher der vertrauenswürdigen Stammzertifizierungsstellen befindet, sind alle Zertifikate, die in seinem Namen ausgestellt werden, sofort vertrauenswürdig. Um neue Zertifikate im Namen Ihres Rootzertifikats auszustellen, geben Sie *makecert.exe* sowohl die *.pvk*- als auch die *.cer*-Datei des Rootzertifikats an.

makecert.exe verwendet nun den öffentlichen Teil Ihres Rootzertifikats aus der *.cer*-Datei und verschlüsselt ihn mit dem geheimen Schlüssel aus der *.pvk*-Datei des Rootzertifikats. Das Ergebnis wird im neuen Zertifikat als Herausgeber gespeichert.

Ab Windows Vista/Windows Server 2008 ist PowerShell auch ohne externe Werkzeuge wie *makecert.exe* in der Lage, selbstsignierte Testzertifikate zu erstellen. Hierbei werden die internen API-Schnittstellen direkt genutzt. Die folgende Funktion *New-CodeSignCert* generiert ein neues Testzertifikat mit einer Gültigkeitsdauer von einem Jahr und installiert es im Zertifikatspeicher des Aufrufers. Wird zusätzlich der Parameter *-CreateTrust* angegeben, kopiert die Funktion das Zertifikat außerdem in die Container der vertrauenswürdigen Stammzertifizierungsstellen und Herausgeber, sodass das Zertifikat anschließend auf dem Computer als vertrauenswürdig gilt. Hierfür sind Administratorrechte erforderlich.

```
# Dank an MVP-Kollege Vadims Podans, der die notwendigen Schnittstellen
# für diese Funktion erforscht und dokumentiert hat
function New-CodeSignCert {
    param (
        [string] [Parameter(Mandatory=$true)] $Name,
        [datetime] $NotBefore = (Get-Date).AddDays(-1),
        [datetime] $NotAfter = $NotBefore.AddDays(365),
        [int] [ValidateSet("1024", "2048")] $KeyLength = 1024,
        [switch] $CreateTrust
    )

    $subject = "CN=$name"

    $OSversion = ([System.Version]((Get-WmiObject Win32_OperatingSystem).Version)).Major
    if ($OSversion -lt 6) {
        Write-Warning "Your operating system version does not support →
            scripted creation of digital certificates."
        return
    }
}
```

```

$SubjectDN = New-Object -ComObject X509Enrollment.CX500DistinguishedName
$SubjectDN.Encode($Subject, 0x0)

$OID = New-Object -ComObject X509Enrollment.CObjectID
$OID.InitializeFromValue("1.3.6.1.5.5.7.3.3")
$OIDs = New-Object -ComObject X509Enrollment.CObjectIDs
$OIDs.Add($OID)

$EKU = New-Object -ComObject X509Enrollment.CX509ExtensionEnhancedKeyUsage
$EKU.InitializeEncode($OIDs)

$PrivateKey = New-Object -ComObject X509Enrollment.CX509PrivateKey
$PrivateKey.ProviderName = "Microsoft Base Cryptographic Provider v1.0"
$PrivateKey.KeySpec = 0x2
$PrivateKey.Length = $KeyLength
$PrivateKey.MachineContext = 0x0
$PrivateKey.Create()

$Cert = New-Object -ComObject X509Enrollment →
    .CX509CertificateRequestCertificate
$Cert.InitializeFromPrivateKey(0x1,$PrivateKey,"")
$Cert.Subject = $SubjectDN
$Cert.Issuer = $Cert.Subject
$Cert.NotBefore = $NotBefore
$Cert.NotAfter = $NotAfter
$Cert.X509Extensions.Add($EKU)
$Cert.Encode()

$Request = New-Object -ComObject X509Enrollment.CX509enrollment
$Request.InitializeFromRequest($Cert)
$endCert = $Request.CreateRequest(0x1)
$Request.InstallResponse(0x2,$endCert,0x1,"")

if ($CreateTrust) {
    [Byte[]]$bytes = [System.Convert]::FromBase64String($endCert)
    foreach ($Container in "Root", "TrustedPublisher") {
        $x509store = New-Object Security.Cryptography.X509Certificates →
            .X509Store $Container, "CurrentUser"
        $x509store.Open([Security.Cryptography.X509Certificates.OpenFlags] →
            ::ReadWrite)
        $x509store.Add([Security.Cryptography.X509Certificates →
            .X509Certificate2]$bytes)
        $x509store.Close()
    }
}

```

Um ein neues Testzertifikat namens *MeinTestCert* zu erstellen und für vertrauenswürdig zu erklären, rufen Sie die Funktion anschließend folgendermaßen auf:

```
PS > New-CodeSignCert -Name MeinTestCert -CreateTrust
```


Prüfen, ob ein Zertifikat vertrauenswürdig ist

Sie möchten feststellen, ob ein bestimmtes Zertifikat vertrauenswürdig ist oder ob es ein Problem mit der Sicherheit eines Zertifikats gibt.

Lösung

Lesen Sie das gewünschte Zertifikat aus dem Zertifikatspeicher aus und rufen Sie dessen Methode `Verify()` auf. Sie liefert *\$true* zurück, wenn das Zertifikat vertrauenswürdig ist, sonst *\$false*. Der folgende Code verwendet das erstbeste Codesign-Zertifikat aus dem persönlichen Zertifikatspeicher:

```
PS > $zertifikat = dir cert:\CurrentUser\My -code | Select-Object -first 1
PS > 'Zertifikat {0} ist vertrauenswürdig? {1}' -f $zertifikat.Subject, →
    $zertifikat.Verify()
Zertifikat CN=PowerShellTestCert ist vertrauenswürdig? False
```

Möchten Sie den Grund erfahren, warum ein Zertifikat nicht vertrauenswürdig ist, benötigen Sie ein `X509Chain`-Objekt, mit dem die Vertrauenskette des Zertifikats überprüft wird:

```
PS > $zertifikat = dir cert:\CurrentUser\My -code | Select-Object -first 1
PS > $chain = New-Object System.Security.Cryptography.X509Certificates.X509Chain
PS > $chain.Build($zertifikat)
False
PS > $chain
ChainContext ChainPolicy ChainStatus ChainElements
-----
138911912 System.Security.C... {System.Security... {System.Securit...
PS > $chain.ChainStatus

Status StatusInformation
-----
UntrustedRoot Eine Zertifikatkette wurde zwar verarbeitet, ende...

PS > $chain.ChainStatus | Select-Object *
Status : UntrustedRoot
StatusInformation : Eine Zertifikatkette wurde zwar verarbeitet, endete
                  jedoch mit einem Stammzertifikat, das beim Vertrau
                  ensanbieter nicht als vertrauenswürdig gilt.
```

Hintergrund

Zertifikate bilden die Grundlage für verlässliche digitale Signaturen, aber weil sich jeder Zertifikate selbst erstellen kann, werden Zertifikate erst dann zu einem wichtigen Sicherheitselement, wenn sie vertrauenswürdig sind.

Vertrauenswürdig heißt: Sie vertrauen dem Aussteller des Zertifikats, weil Sie ihn kennen (zum Beispiel Ihre eigene PKI) oder weil es sich um ein allgemein vertrauenswürdiges Zertifizierungsunternehmen handelt.

Selbstsignierte Zertifikate sind grundsätzlich zunächst nicht vertrauenswürdig, und das Beispiel zeigt auch, warum das so ist: »Eine Zertifikatkette wurde zwar verarbeitet, endete jedoch mit einem Stammzertifikat, das beim Vertrauensanbieter nicht als vertrauenswürdig gilt.« Übersetzt heißt das: Der Herausgeber des selbstsignierten Zertifikates ist keine besonders vertrauenswürdige Autorität und deshalb ist auch das Zertifikat selbst nicht vertrauenswürdig.

Sie müssen selbstsignierte Zertifikate deshalb zunächst in den besonderen Zertifikatspeicher der vertrauenswürdigen Stammzertifizierungsstellen aufnehmen, um es vertrauenswürdig zu machen. Dieser Schritt ist nur bei selbstsignierten Zertifikaten erforderlich. Wurde das Zertifikat von einer anderen Autorität ausgestellt (zum Beispiel einer Zertifizierungsstelle oder einem eigenen Rootzertifikat), muss stattdessen gewährleistet sein, dass sich das Rootzertifikat im Zertifikatspeicher der vertrauenswürdigen Stammzertifizierungsstellen befindet.

Selbstsigniertes Zertifikat für vertrauenswürdig erklären

Sie möchten ein Zertifikat für vertrauenswürdig erklären, das Sie sich selbst ausgestellt haben, zum Beispiel mit dem Werkzeug *makecert.exe*.

Lösung

Kopieren Sie das Zertifikat in den Zertifikatspeicher der vertrauenswürdigen Stammzertifizierungsstellen. Hierfür sind Administratorrechte erforderlich. Der folgende Code kopiert ein Zertifikat namens *PowerShellTestCert* aus Ihrem persönlichen Zertifikatspeicher in den Speicher für vertrauenswürdige Stammzertifizierungsstellen (namens *root*):

```
PS > $zert = dir cert:\CurrentUser\My -codeSign | ? { $_.Subject -eq 'CN=PowerShellTestCert' } | Select-Object -first 1
PS > $zert

Verzeichnis: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                     Subject
-----
44F3D50ED03FD846ED11FC5295D100DA736B1D01  CN=PowerShellTestCert

PS > $Store = New-Object system.security.cryptography.X509Certificates.X509Store ->
("Root", "CurrentUser")
PS > $Store.Open("ReadWrite")
PS > $Store.Add($zert)
PS > $Store.Close()
PS > $zert.Verify()
True
```

ACHTUNG Der Zertifikatimport erfolgt dabei einwandfrei, doch erscheint das importierte Zertifikat möglicherweise nicht im Zertifikatspeicher. Auch alle Vertrauensprüfungen scheitern. Der Grund: Bei einigen Windows Vista-Versionen gibt es eine fehlerhafte Berechtigungseinstellung für den folgenden Registrierungsschlüssel:

HKEY_CURRENT_USER\Software\Microsoft\SystemCertificates\Root\ProtectedRoots

Dieser Schlüssel muss für den aktuellen Benutzer Leserechte gewähren. Bei einigen OEM-Installationen von Windows Vista ist das nicht der Fall. Sie beheben das Problem, indem Sie dem angegebenen Schlüssel für den aktuellen Benutzer Leserechte einräumen. Das genaue Vorgehen ist hier beschrieben:

<http://support.microsoft.com/default.aspx/kb/932156/en-us>

Hintergrund

Selbstsignierte Zertifikate haben eine Besonderheit: Deren Herausgeber ist mit ihnen selbst identisch. Das Zertifikat ist also gleichzeitig sein eigener Herausgeber.

```
PS > $zert = dir cert:\ -recurse -codeSign | ? { $_.Subject -like →  
    'CN=PowerShellTest*' } | Select-Object -first 1  
PS > $zert.Subject  
CN=PowerShellTestCert  
PS > $zert.Issuer  
CN=PowerShellTestCert  
PS > if ($zert.Subject -eq $zert.Issuer) { "selbstsigniert" } else →  
    { "Herausgeber: $($zert.Issuer)" }  
selbstsigniert
```

Damit selbstsignierte Zertifikate vertrauenswürdig sind, muss eine Kopie im Zertifikatspeicher der vertrauenswürdigen Stammzertifizierungsstellen hinterlegt werden. Das können Sie manuell im Zertifikat-Manager (*certmgr.msc*) per Drag & Drop oder programmgesteuert über .NET Framework tun.

Fügen Sie das Zertifikat dann in den gewünschten Zertifikatspeicher ein. Eine Aufstellung der internen Namen für die Zertifikatspeicher liefert die Tabelle 13.2. Der interne Name für den Zertifikatspeicher der vertrauenswürdigen Stammzertifizierungsstellen lautet *Root*. Eine Liste sämtlicher verfügbarer Zertifikatspeichernamen liefert die Aufzählung *X509Certificates.StoreName*:

```
PS > [System.Enum]::GetNames([System.Security.Cryptography.X509Certificates.StoreName])  
AddressBook  
AuthRoot  
CertificateAuthority  
Disallowed  
My  
Root  
TrustedPeople  
TrustedPublisher
```

Speicher	Name	Beschreibung
Eigene Zertifikate	<i>My</i>	Speichert Ihre persönlichen Zertifikate dir cert:\currentuser\my
Vertrauenswürdige Stammzertifizierungsstellen	<i>Root</i>	Stammzertifikate von Zertifizierungsstellen. Alle Zertifikate, die von diesen Zertifizierungsstellen ausgegeben werden, gelten automatisch als vertrauenswürdig. dir cert:\currentuser\root
Organisationsvertrauen	<i>Trust</i>	Stammzertifikate von Zertifizierungsstellen. Alle Zertifikate, die von diesen Zertifizierungsstellen ausgegeben werden, gelten automatisch als vertrauenswürdig. dir cert:\currentuser\trust
Zwischenzertifizierungsstellen	<i>CA</i>	Stammzertifikate von Zertifizierungsstellen. Alle Zertifikate, die von diesen Zertifizierungsstellen ausgegeben werden, gelten automatisch als vertrauenswürdig. dir cert:\currentuser\ca
Active Directory-Benutzerobjekte	<i>UserDS</i>	Zertifikate von Active Directory dir cert:\currentuser\usersds
Vertrauenswürdige Herausgeber	<i>TrustedPublisher</i>	Zertifikate von Herausgebern, denen Sie vertrauen dir cert:\currentuser\trustedpublisher
Nicht vertrauenswürdige Zertifizierungsstellen	<i>Disallowed</i>	Sperrliste mit Zertifikaten von Zertifizierungsstellen, denen auf keinen Fall vertraut wird dir cert:\currentuser\disallowed
Drittanbieter-Stammzertifizierungsstellen	<i>AuthRoot</i>	Stammzertifikate von Zertifizierungsstellen. Alle Zertifikate, die von diesen Zertifizierungsstellen ausgegeben werden, gelten automatisch als vertrauenswürdig. dir cert:\currentuser\authroot
Vertrauenswürdige Personen	<i>TrustedPeople</i>	Zertifikate von Personen, denen Sie vertrauen dir cert:\currentuser\trustedpeople
Smartcard vertrauenswürdige Stämme	<i>SmartCardRoot</i>	Stammzertifikate für Smartcard-Zertifikate dir cert:\currentuser\smartcardroot

Tabelle 13.2 Zertifikatspeicher des aktuellen Benutzers

PowerShell-Skript signieren

Sie möchten ein PowerShell-Skript mit einer digitalen Signatur versehen, um beispielsweise sicherzustellen, dass das Skript nicht nachträglich verändert wird, oder um die erhöhten Sicherheitseinstellungen der ExecutionPolicy nutzen zu können.

Lösung

Verwenden Sie *Set-AuthenticodeSignature*, um einem PowerShell-Skript eine digitale Signatur hinzuzufügen.

Liegt das Zertifikat als *.pfx*-Datei vor, laden Sie das Zertifikat folgendermaßen:

```
PS > $pfx = "C:\pfx\my_cert.pfx"
PS > $zert = Get-PfxCertificate $pfx
```

Möchten Sie lieber auf ein bereits installiertes Zertifikat zugreifen, verwenden Sie das Laufwerk cert:. Die folgende Zeile liefert das erstbeste Codesigning-Zertifikat aus dem persönlichen Zertifikatspeicher:

```
PS > $zert = Get-ChildItem cert:\CurrentUser\My -codesign | Select-Object -first 1
```

Anschließend können Sie eine oder mehrere Dateien damit signieren:

```
PS > # eine einzelne Datei signieren:
PS > Set-AuthenticodeSignature $home\skript.ps1 $zert
PS > alle Skriptdateien im eigenen Profil rekursiv signieren:
PS > Dir $home -filter *.ps1 -recurse | Set-AuthenticodeSignature -cert $zert
```

Wenn Sie mehrere Codesigning-Zertifikate besitzen, könnten Sie auch per Dialogfeld eines davon für die Signatur auswählen. Der folgende Code würde sämtliche PowerShell-Skripts in Ihrem Profil mit einer neuen digitalen Signatur versehen. Die zur Signatur verwendete Identität könnten Sie sich zuvor in einem Dialogfeld aussuchen:

```
PS > $titel = "Verfügbare Identitäten"
PS > $text = "Bitte Zertifikat für Signatur auswählen"
PS >
PS > $zertifikate = dir cert:\CurrentUser\My -codesigning
PS > [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > $container = New-Object System.Security.Cryptography.X509Certificates →
    .X509Certificate2Collection
PS > $zertifikate | Foreach-Object { [void]$container.Add($_) }
PS >
PS > $zertifikat = [System.Security.Cryptography.x509Certificates →
    .X509Certificate2UI]::SelectFromCollection($container, $titel, $text, 0)
PS > Dir $home -filter *.ps1 -recurse | Set-AuthenticodeSignature -certificate →
    $zertifikat[0]
```

HINWEIS

PowerShell signiert Dateien nur, wenn Sie ein gültiges (vertrauenswürdigen) Codesigning-Zertifikat angeben. Außerdem müssen die Dateien eine Mindestgröße von 4 Byte besitzen und in einem Standardformat gespeichert sein. Speichern Sie Skriptdateien beispielsweise im Format »Unicode Big Endian«, kann *Set-AuthenticodeSignature* diese Dateien nicht signieren – leider speicherten die ersten Versionen des PowerShell ISE-Editors genau in diesem Format. In allen diesen Fällen bekommen Sie die Meldung »UnknownError« zurück-

geliefert. *Set-AuthenticodeSignature* kann nicht nur Skriptdateien signieren. Auch PowerShell-Moduldateien (*.psm1) sowie alle sonstigen grundsätzlich signierbaren Dateitypen (*.exe, *.dll, etc.) lassen sich damit bearbeiten. Es ist nicht möglich, einer Datei mehr als eine digitale Signatur zuzuweisen (Co-Signing). Existiert bereits eine Signatur, wird sie von der neuen Signatur überschrieben.

Hintergrund

Damit Sie ein PowerShell-Skript signieren können, benötigen Sie ein Zertifikat, für das Sie über einen privaten Schlüssel verfügen und das den Verwendungszweck *Codesignatur* trägt. In der folgenden Zeile wird beispielsweise ein Zertifikat aus dem persönlichen Zertifikatspeicher gewählt, dessen Name mit »PowerSh« beginnt:

```
PS > $zert = dir cert:\CurrentUser\My -codeSign | ? { $_.Subject -like →
'CN=PowerSh*' }
PS > $zert
```

Verzeichnis: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint	Subject
8475C5C5DE9ACF4E3A2F320E572FAFF672876399	CN=PowerShellTestCert

Alternativ kann ein Zertifikat auch mit *Get-PfxCertificate* aus einer pfx-Datei geladen werden, sodass das Zertifikat, das Sie für eine Signatur einsetzen wollen, nicht auf dem Computer installiert zu sein braucht. Anschließend übergeben Sie *Set-AuthenticodeSignature* den Pfadnamen zu Ihrer Skriptdatei sowie das Zertifikat, mit dem das Skript signiert werden soll:

```
PS > Set-AuthenticodeSignature $home\skript.ps1 $zert
```

Verzeichnis: C:\Users\Tobias

SignerCertificate	Status	Path
8475C5C5DE9ACF4E3A2F320E572FAFF672876399	Valid	skript.ps1

Die digitale Unterschrift wird in Form eines Kommentarblocks in die Skriptdatei eingefügt.

```
PS > notepad $home\skript.ps1
```

Bei der Signatur handelt es sich um einen Hash (einen digitalen Fingerabdruck) des Skriptcodes, der mit dem privaten Schlüssel des Zertifikats verschlüsselt ist. Mithilfe dieser Signatur kann man nun zweifelsfrei feststellen, ob das Skript nachträglich (nach der Signatur) verändert wurde und wer das Skript digital unterschrieben hat.

Die signierten Skripts gelten bei dieser Form der Signatur nur so lange als gültig signiert, wie das zugrundeliegende Zertifikat gültig ist. Das kann in der Praxis große Folgeprobleme verursachen. Da die Gültigkeit eines Zertifikats eigentlich nur beim Zeitpunkt der Signatur maßgeblich ist, kann man digitale Signaturen auch mit einem sogenannten Timestamp versehen. Hierbei wird während des Signaturvorgangs eine Verbindung zu einem autorisierten Timestamp-Server hergestellt. Er bestätigt, dass das Zertifikat zu diesem Zeitpunkt gültig ist. So wird verhindert, dass die Signatur ihre Gültigkeit verliert, wenn das zugrundeliegende Zertifikat abläuft.

Die folgende Zeile signiert sämtliche Skripts in Ihrem Profilordner und greift dabei auf einen öffentlichen Timestamp-Server zu. Die Signaturen behalten auf diese Weise ihre Gültigkeit, auch wenn das zugrundeliegende Zertifikat abgelaufen ist. Für den Signierungsvorgang ist eine Internetverbindung erforderlich:

```
PS > $zert = Get-ChildItem cert:\CurrentUser\My -codeSign | Select-Object -first 1
PS > Dir $home -filter *.ps1 -recurse | Set-AuthenticodeSignature -cert $zert →
-TimestampServer 'http://timestamp.verisign.com/scripts/timestamp.dll'
```

Skriptsignatur überprüfen

Sie möchten die digitale Signatur eines Skripts überprüfen, zum Beispiel, um festzustellen, ob das Skript seit der Signatur verändert wurde, oder um herauszufinden, wer das Skript seinerzeit signiert hat.

Lösung

Verwenden Sie *Get-AuthenticodeSignature*, um die Signatur zu lesen.

```
PS > $signatur = Get-AuthenticodeSignature $home\skript.ps1
PS > $signatur

Verzeichnis: C:\Users\Tobias
```

SignerCertificate	Status	Path
-----	-----	----
DCA1B3CC380EC20D04D50320A0715DEF379FC8F0	Valid	skript.ps1

```
PS > $signatur.StatusMessage
Signatur wurde überprüft.
```

In der Eigenschaft *SignerCertificate* finden Sie das Zertifikat, mit dem diese digitale Unterschrift geleistet wurde. So finden Sie heraus, wer unterschrieben hat:

```
PS > [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Security")
PS > [System.Security.Cryptography.X509Certificates.X509Certificate2UI] →
::DisplayCertificate( $signatur.SignerCertificate)
```

Möchten Sie mehrere Dateien überprüfen, leiten Sie das Ergebnis von *Get-ChildItem* weiter an *Get-AuthenticodeSignature*. Die folgende Zeile überprüft alle Skripts in Ihrem persönlichen Profilordner:

```
PS > Get-ChildItem $home -filter *.ps1 -recurse | Get-AuthenticodeSignature
```

Möchten Sie nur Skripts finden, die keine gültige Signatur besitzen, filtern Sie das Ergebnis nach *Status*:

```
PS > Get-ChildItem $home -filter *.ps1 -recurse | Get-AuthenticodeSignature | →  
Where-Object { $_.Status -ne 'Valid' }
```

Hintergrund

Get-AuthenticodeSignature liest die Signatur aus einem PowerShell-Skript und liefert ein *Signature*-Objekt zurück. Dessen wichtigste Information findet sich in der Eigenschaft *Status*. Ist eine Signatur vorhanden und gültig, und wurde der Skriptcode seit der Signatur nicht verändert, lautet das Ergebnis *Valid*. Alle übrigen Meldungen und ihre Ursachen listet Tabelle 13.3 auf.

Status	Meldung	Beschreibung
<i>NotSigned</i>	Die Datei xyz ist nicht digital signiert. Das Skript wird auf dem System nicht ausgeführt. Weitere Informationen erhalten Sie mit <i>get-help about_signing</i> .	Die Datei enthält keine digitale Signatur. Signieren Sie die Datei mit <i>Set-AuthenticodeSignature</i> .
<i>UnknownError</i>	Eine Zertifikatkette wurde zwar verarbeitet, endete jedoch mit einem Stammzertifikat, das beim Vertrauensanbieter nicht als vertrauenswürdig gilt	Das verwendete Zertifikat ist unbekannt. Fügen Sie den Herausgeber des Zertifikats in den Speicher für vertrauenswürdige Stammzertifizierungsstellen ein.
<i>HashMismatch</i>	Der Inhalt der Datei »...« wurde möglicherweise manipuliert, da der Hash der Datei nicht mit dem in der digitalen Signatur gespeicherten Hash übereinstimmt. Das Skript wird auf dem System nicht ausgeführt. Weitere Informationen erhalten Sie mit <i>get-help about_signing</i> .	Der Inhalt der Datei wurde verändert. Wenn Sie selbst den Inhalt verändert haben, signieren Sie die Datei neu.
<i>Valid</i>	Signatur wurde überprüft	Der Dateiinhalt stimmt mit der Signatur überein, und die Signatur ist gültig

Tabelle 13.3 Statusmeldungen der Signaturüberprüfung und ihre Ursachen

Zertifikat als .cer-Datei exportieren

Sie möchten den öffentlichen Teil eines Zertifikats als .cer-Datei exportieren, zum Beispiel, um das Zertifikat zu sichern, auf einen anderen Computer zu übertragen oder unternehmensweit zu verteilen.

Lösung

Sprechen Sie das Zertifikat, das Sie exportieren möchten, an, und rufen Sie dessen *Export()*-Methode auf. Speichern Sie das Ergebnis des Exportvorgangs in einer Datei. Die folgenden Zeilen exportieren ein Zertifikat namens *PowerShellTestCert* in der Datei *zertifikat.cer*:

```
PS > $filename = "$home\zertifikat.cer"
PS > $zertname = 'PowerShellTestCert'
PS > $zert = dir cert:\ -recurse -codeSign | ? { $_.Subject -eq "CN=$zertname" } →
    | Select-Object -first 1
PS > $bytes = $zert.Export('Cert')
PS > $filestream = New-Object System.IO.FileStream($filename, 'Create')
PS > $filestream.Write($bytes, 0, $bytes.Length)
PS > $filestream.Close()
PS > dir $filename
```

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	17.04.2008	13:05	546 zertifikat.cer

Hintergrund

Jedes Zertifikat besitzt eine *Export()*-Methode, mit der man das Zertifikat exportieren kann. Dazu geben Sie *Export()* lediglich an, in welchem Format das Zertifikat exportiert werden soll. Um den öffentlichen Teil des Zertifikats zu exportieren, lautet die korrekte Angabe »Cert«. Die übrigen möglichen Formate liefert die Aufzählung *X509Certificates.X509ContentType*:

```
PS > [System.Enum]::GetNames([System.Security.Cryptography →
    .X509Certificates.X509ContentType])
Unknown
Cert
SerializedCert
Pfx
Pkcs12
SerializedStore
Pkcs7
Authenticode
```

Das Ergebnis von *Export()* ist der binäre Inhalt des Zertifikats. Dieser muss anschließend mit einem *FileStream*-Objekt in eine Datei geschrieben werden. Das Ergebnis ist eine .cer-Datei, mit

der Sie das exportierte Zertifikat jederzeit wieder neu im Zertifikatspeicher des Computers installieren können:

```
PS > . $filename
```

Möchten Sie das Zertifikat unternehmensweit verteilen, verwenden Sie die Gruppenrichtlinien.

Zertifikat als .pfx-Datei exportieren

Sie möchten ein Zertifikat mitsamt seinem (geheimen) privaten Schlüssel exportieren und in einer .pfx-Datei speichern, zum Beispiel, um eine Sicherheitskopie eines wichtigen Zertifikats anzulegen oder das Zertifikat anschließend auf einem anderen Rechner zu installieren.

Lösung

Greifen Sie auf das gewünschte Zertifikat zu und exportieren Sie es mit dessen *Export()*-Methode als .pfx-Datei. Der folgende Code exportiert das Zertifikat *PowerShellTestCert* in die Datei *zertifikat.pfx*:

```
PS > $filename = "$home\zertifikat.pfx"
PS > $zertname = 'PowerShellTestCert'
PS > $kennwort = 'strenggeheim'
PS >
PS > $zert = dir cert:\ -recurse -codeSign | ? { $_.Subject -eq "CN=$zertname" } →
    | Select-Object -first 1
PS >
PS > $bytes = $zert.Export('Pfx', $kennwort)
PS > $filestream = New-Object System.IO.FileStream($filename, 'Create')
PS > $filestream.Write($bytes, 0, $bytes.Length)
PS > $filestream.Close()
PS > dir $filename
```

Verzeichnis: Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	01.02.2011 15:29	1740	zertifikat.pfx

Hintergrund

Zu jedem Zertifikat gehört ein geheimer Schlüssel. Nur der Eigentümer eines Zertifikats darf den geheimen privaten Schlüssel kennen, denn wer darauf Zugriff hat, kann mit dem Zertifikat digitale Unterschriften leisten.

Die *Export()*-Methode des Zertifikats kann den Inhalt des Zertifikats in verschiedenen Formaten exportieren. Dazu zählt auch das Format *Pfx*. In diesem Format wird nicht nur der öffentliche Teil des Zertifikates gespeichert, sondern außerdem auch sein privater Schlüssel. Damit ist es

möglich, aus einer *.pfx*-Datei das gesamte Zertifikat einschließlich privatem Schlüssel wiederherzustellen, und deshalb sind *.pfx*-Dateien die beste Wahl, eigene Zertifikate als Backup zu sichern.

Ob ein Zertifikat über einen privaten Schlüssel verfügt, verrät die Eigenschaft *hasPrivateKey*. Die folgende Zeile listet sämtliche Zertifikate auf, die über einen privaten Schlüssel verfügen:

```
PS > dir cert:\ -recurse | ? { $_.hasPrivateKey -eq $true } | Select-Object -unique
```

Verzeichnis: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint	Subject
-----	-----
DCA1B3CC380EC20D04D50320A0715DEF379FC8F0	CN=Mitarbeiter1
D85EFBE4808E9CF97873940492FD1EFE55D2B330	CN=Tobias
8475C5C5DE9ACF4E3A2F320E572FAFF672876399	CN=PowerShellTestCert

Weil die *.pfx*-Datei den sicherheitskritischen privaten Schlüssel enthält, wird die *.pfx*-Datei beim Exportvorgang außerdem mit einem symmetrischen Schlüssel (einem Kennwort) verschlüsselt, den Sie der Methode *Export()* als zweiten Parameter übergeben. Man kann die *.pfx*-Datei später nur dann wiederherstellen, wenn man dieses Kennwort kennt.

Zertifikat löschen

Sie möchten ein installiertes Zertifikat aus dem Zertifikatspeicher löschen, weil Sie es nicht länger benötigen.

ACHTUNG Wenn Sie ein Zertifikat löschen, ist es verloren. Sie können es nur dann wiederherstellen, wenn Sie es vorher in einer Sicherungsdatei exportiert haben. Löschen Sie ein Zertifikat, verlieren Sie gegebenenfalls die Möglichkeit, mit seiner Hilfe verschlüsselte Daten wiederherzustellen.

Lösung

Zertifikate können von Windows PowerShell nur indirekt über .NET Framework gelöscht werden. Wenn Sie Speicherort und Thumbprint des Zertifikats kennen, gehen Sie so vor:

```
PS > # Zu löschendes Zertifikat besorgen
PS > $zertifikat = Get-Item cert:\currentuser\my\B0466FBBA3E1830C922FF9 →
A722EFA3D1F6C52BA6

PS > # Zertifikatspeicher schreibend öffnen und Zertifikat löschen:
PS > $store = New-Object System.Security.Cryptography.X509Certificates.X509Store →
"My", "CurrentUser"
PS > $store.Open("ReadWrite")
PS > $store.Remove($zertifikat)
PS > $store.Close()
```

Möchten Sie das Zertifikat aufgrund einer anderen Eigenschaft auswählen, zum Beispiel über seinen Namen, gehen Sie so vor:

```
PS > # Zu löschende Zertifikate auswählen
PS > $zertifikate = dir cert:\ -Recurse | Where-Object { $_.Subject -eq 'CN=TobiasWeltner' }

PS > # Zertifikatspeicher schreibend öffnen und Zertifikat löschen:
PS > $store = New-Object System.Security.Cryptography.X509Certificates.X509Store "My", "CurrentUser"
PS > $store.Open("ReadWrite")
PS > $zertifikate | ForEach-Object { $store.remove($_) }
PS > $store.Close()
```

Gelöscht werden dabei jeweils nur Zertifikate, die in dem zum Schreiben geöffneten Zertifikatspeicher liegen. Im Beispiel ist dies *CurrentUser\My*.

Hintergrund

Normalerweise sollte PowerShell selbst in der Lage sein, Zertifikate mit *Remove-Item* zu entfernen, und die PowerShell-Hilfe erklärt ausdrücklich, dass *Remove-Item* hierzu in der Lage ist. Dennoch scheitert dieser Ansatz:

```
PS > Remove-Item cert:\currentuser\my\8475C5C5DE9ACF4E3A2F320E572FAFF672876399
Remove-Item : Die Ausführung des Anbieters wurde beendet, da der Anbieter diesen Vorgang nicht unterstützt.
Bei Zeile:1 Zeichen:12
+ Remove-Item <<<< cert:\currentuser\my\8475C5C5DE9ACF4E3A2F320E572FAFF672876399
```

Tatsächlich kann PowerShell nur lesend auf den Zertifikatspeicher zugreifen. Damit ist es *Remove-Item* unmöglich, Änderungen am Zertifikatspeicher vorzunehmen, und Sie müssen auf Low-Level-Methoden von .NET Framework zurückgreifen. .NET Framework verlangt von Ihnen, dass Sie den passenden Zertifikatspeicher öffnen, also denjenigen, in dem sich das zu löschende Zertifikat befindet. Persönliche Zertifikate befinden sich stets im Speicher *CurrentUser\My*. Wenn Sie andere Zertifikate löschen wollen, muss der Zertifikatspeicher im Code angepasst werden.

Da jedes Zertifikat in *PSParentPath* den Namen seines Zertifikatspeichers nennt, könnte man eine eigene Funktion namens *Remove-Cert* nachrüsten:

```
Filter Remove-Cert {  
    $path = $_.PSParentPath  
    $storename = Split-Path $path -leaf  
    $folder = Split-Path $path -parent  
    $folder = Split-Path $folder -noQualifier  
  
    $store = New-Object System.Security.Cryptography.X509Certificates.X509Store →  
        $storename,$folder  
    $store.Open("ReadWrite")  
    $store.Remove($_)  
    $store.Close()  
}
```

So sind Sie in der Lage, beliebige Zertifikate aus beliebigen Zertifikatspeichern zu löschen (vorausgesetzt, Sie verfügen über die dafür nötigen Zugriffsrechte). Der folgende Code findet alle Zertifikate, in deren Name das Wort »test« vorkommt, und löscht sie in allen Zertifikatspeichern. Auf diese Weise können Sie selbstsignierte Testzertifikate einschließlich ihrer Kopien im Ordner der vertrauenswürdigen Stammzertifizierungsstellen und -Herausgeber sauber entfernen.

```
PS > $zertifikate = dir cert:\ -recurse | Where-Object { $_.Subject -like →  
    '*test*' }  
PS > # Zertifikatspeicher schreibend öffnen und Zertifikat löschen:  
PS > $zertifikate | Remove-Cert
```

ACHTUNG Einmal gelöschte Zertifikate können Sie nur dann wiederherstellen, wenn Sie über eine Sicherungskopie verfügen. Seien Sie beim Löschen also extrem vorsichtig. Entfernen Sie die falschen Zertifikate, verlieren Sie Zugriff auf verschlüsselte Informationen wie zum Beispiel verschlüsselte E-Mail-Nachrichten oder das verschlüsselnde Dateisystem.

Zusammenfassung

PowerShell enthält eine vollständige Unterstützung für den Umgang mit digitalen Signaturen. Mit *Get-AuthenticodeSignature* kann man die digitale Signatur von Skripts (und anderen signierten Dateien) überprüfen. Dies funktioniert auch rekursiv, sodass sich mit einer einzelnen Zeile Code beispielsweise sämtliche Skripts eines sicherheitskritischen Servers prüfen lassen.

Mit *Set-AuthenticodeSignature* lassen sich Skripts (und andere signierbare Dateien) mit einer neuen digitalen Signatur versehen. Dies funktioniert ebenfalls rekursiv, sodass sich erneut mit nur einer Zeile Code sämtliche Skripts eines Systems mit digitalen Signaturen versehen, quasi »versiegeln« lassen. Sobald ein Skript eine digitale Signatur trägt, kann jederzeit überprüft werden, ob die Datei noch im Originalzustand vorliegt oder geändert wurde. Auch die Identität der Person, die das Skript ursprünglich signiert hat (und also verantwortet), ist lesbar.

Diese Mechanismen funktionieren bereits ohne weitere Konfigurationsarbeit oder Änderungen an den Sicherheitseinstellungen. Über die *ExecutionPolicy* kann man darüber hinaus festlegen,

dass PowerShell die Signaturen von PowerShell-Skripts automatisch überprüft. Ist die digitale Signatur eines Skripts ungültig oder nicht vorhanden, kann die ExecutionPolicy dafür sorgen, dass das Skript nicht mehr ausführbar ist. Ob eine solche automatische Prüfung notwendig ist, hängt vom jeweiligen Fall ab. Auf einem sicherheitskritischen Produktionsserver kann man so allerdings wirksam verhindern, dass Unbefugte Änderungen an Skripts vornehmen. Die Änderungen selbst verhindert die ExecutionPolicy zwar nicht, doch sind die Skripts danach nicht mehr ausführbar.

TIPP

Noch weitreichendere Kontrolle bieten die Gruppenrichtlinieneinstellungen im Bereich der *Richtlinien für Softwareeinschränkung*, die ebenfalls auf Signaturen reagieren können. Hier kann man festlegen, dass PowerShell-Skripts nur mit einer bestimmten digitalen Signatur ausführbar sein sollen. Die ExecutionPolicy dagegen erlaubt die Ausführung bei jeder beliebigen gültigen digitalen Signatur.

Grundlage sämtlicher Signaturvorgänge sind digitale Zertifikate, die den Verwendungszweck »Codesigning« beinhalten müssen und für die ein privater Schlüssel vorliegen muss. PowerShell kann mit seinem virtuellen Laufwerk *cert:* auf den Windows-Zertifikatspeicher zugreifen und so mit allen installierten Zertifikaten zusammenarbeiten. Darüber hinaus lassen sich Zertifikate mit *Get-PfxCertificate* auch vorübergehend aus *.pfx*-Dateien einlesen.

Über die Low-Level-Methoden von .NET Framework kann PowerShell noch weitreichendere Kontrolle ausüben und zum Beispiel Zertifikate generieren, automatisiert installieren, importieren oder exportieren.

Kapitel 14

XML-Daten verarbeiten

In diesem Kapitel:

Auf den Inhalt einer XML-Datei zugreifen	408
Mit XPath Informationen in XML finden	413
XML-Informationen filtern	417
Neues XML-Dokument erstellen	418
Neues XML-Dokument aus Schablone erstellen	420
Inhalt einer XML-Datei ändern	426
Zusammenfassung	428

Während früher Informationen häufig in Form sogenannter INI-Dateien gespeichert wurden, übernimmt heute vermehrt das XML-Format (eXtended Markup Language) diese Rolle. XML ist eine Datenbeschreibungssprache, die Ähnlichkeiten zu HTML aufweist. Wegen der umfangreichen und teils stark verschachtelten Informationen ist es nicht immer leicht, Informationen aus XML zu lesen oder selbst in XML zu speichern. PowerShell erleichtert deshalb den Zugang zu XML über eine Reihe von Cmdlets und Datentypen.

In diesem Kapitel erfahren Sie, wie Sie Informationen aus XML-Dateien lesen, vorhandene XML-Dateien ändern und eigene Informationen in neuen XML-Dateien speichern.

Auf den Inhalt einer XML-Datei zugreifen

Sie möchten auf den Inhalt einer XML-Datei zugreifen, um zum Beispiel bestimmte Informationen daraus auszulesen.

Lösung

Legen Sie ein neues *XML*-Objekt an und laden Sie den Inhalt der XML-Datei mit der Methode *Load()*. Der XML-Inhalt darf zum Beispiel aus dem Internet stammen und kann ein RSS-Ticker sein:

```
PS > $xml = New-Object XML
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
PS > $xml
```

```
xml                                rss
---                                ---
version="1.0" encoding="ISO-8859-1" ... rss
```

Oder Sie geben den Pfadnamen zu einer vorhandenen XML-Datei an. Im folgenden Beispiel wird die Datei *starter.xml* aus dem Windows-Ordner gelesen, die Informationen für die unbeaufsichtigte Installation von Windows 7 enthält:

```
PS > $xml = New-Object xml
PS > $xml.load("$env:windir\starter.xml")
PS > $xml
```

```
xml                                unattend
---                                -
version="1.0"                      unattend
```

Geben Sie den Inhalt des XML-Objekts aus, zeigt es jeweils die obersten Knoten der XML-Datei. In diese können Sie sich danach schrittweise hineintasten, um die im XML enthaltenen Informationen zu lesen:


```

PS > $xml.unattend

settings
-----
settings

PS > $xml.unattend.settings

pass                                component
----                                -
offlineServicing                   {Microsoft-Windows-BaseCrashDumpSett...

PS > $xml.unattend.settings.component

name                                : Microsoft-Windows-BaseCrashDumpSettings
processorArchitecture               : x86
publicKeyToken                     : 31bf3856ad364e35
language                           : neutral
versionScope                       : nonSxS
#comment                           :

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl]

AutoReboot                         : 1
CrashDumpEnabled                   : 2

name                                : Microsoft-Windows-BaseCrashDumpSettings
processorArchitecture               : amd64
publicKeyToken                     : 31bf3856ad364e35
language                           : neutral
versionScope                       : nonSxS
#comment                           :

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\CrashControl]
(...)

```

Greifen Sie auf RSS-Ticker im Internet zu, ist das ebenso. Tasten Sie sich durch die Knoten, bis Sie zu den Schlagzeilen gelangen:

```

PS > $xml = New-Object XML
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
PS > $xml

xml                                rss
---                                ---
version="1.0" encoding="ISO-8859-1" ... rss

PS > $xml.rss.channel.item | Select-Object Title, Category

```

title	category
-----	-----
Dioxin-Schweine: Verdächtiges Fleisch...	Wirtschaft
Vatikan: Ex-Papst Johannes Paul II. ...	Panorama
Achilles' Ferse: Kavaliere unerwünscht	Sport
Mercedes Benz 219, Baujahr 1959: Tan...	Auto
(...)	

Hintergrund

XML-Dateien enthalten einfachen Text, allerdings eingebettet in eine klare Struktur ähnlich den Tags einer HTML-Seite. Genau genommen beschreibt XML damit die Informationen in einer hierarchisch eindeutigen Weise. XML-Objekte sind auf diesen Datentyp spezialisiert und erleichtern den Zugang zu den darin enthaltenen Informationen.

Ein interessantes Anschauungsobjekt sind RSS-Feeds aus dem Internet, die viele Webseiten mittlerweile anbieten. Dahinter steckt lediglich XML, das die Schlagzeilen des Tages oder andere Informationen verpackt. Mit RSS-Feedreadern kann man aus diesen Informationen die Schlagzeilen und zugehörigen Webseiten anzeigen.

Der RSS-Feed des Nachrichtenmagazins »Der Spiegel« steht beispielsweise unter der folgenden URL zur Verfügung: <http://www.spiegel.de/schlagzeilen/index.rss>. Mit einem XML-Objekt kann der Inhalt gelesen werden:

```
PS > $xml = New-Object XML
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
```

Der Rohinhalt der XML-Datei kann auf folgende Weise sichtbar gemacht werden:

```
PS > $xml.InnerXml
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?> <rss xmlns:content="http://
purl.org/rss/1.0/modules/content/" version="2.0"><channel> <title>SPIEGEL ONLINE -
Schlagzeilen</title><link>http://www.spiegel.de</link><description>Deutschlands führende
Nachrichtenseite. Alles Wichtige aus Politik, Wirtschaft, Sport, Kultur, Wissenschaft,
Technik und mehr.</description><language>de</language><pubDate>Fri, 14 Jan 2011 13:01:24
+0100</pubDate><lastBuildDate>Fri, 14 Jan 2011 13:01:24 +0100</
lastBuildDate><image><title>SPIEGEL ONLINE</title><link>http://www.spiegel.de/</
link><url>http://www.spiegel.de/static/sys/logo_120x61.gif</url></
image><item><title>Dioxin-Schweine: Verdächtiges Fleisch landete auch in Polen und
Tschechien </title><link>http://www.spiegel.de/wirtschaft/service...
```

Das XML-Objekt liefert die obersten Knoten direkt zurück, also in diesem Fall XML und RSS:

```
PS > $xml

xml                                rss
---                                ---
version="1.0" encoding="ISO-8859-1" ... rss
```

Sie können nun stufenweise in das XML-Objektmodell einsteigen:

```
PS > $xml.rss
```

content	version	channel
-----	-----	-----
http://purl.org/rss/1.0...	2.0	channel

Allerdings kommt es zu einem Problem, wenn Sie noch eine Stufe tiefer steigen und *channel* auflisten wollen:

```
PS > $xml.rss.channel
```

```
format-default : Der Member "Item" ist bereits vorhanden.
```

Der Grund: Der Knoten *channel* enthält einen Unterknoten namens *item*. »item« ist aber ausnahmsweise ein reserviertes Wort, das PowerShell nicht als neue Eigenschaft dem Objekt hinzufügen kann. Dieses Problem tritt also bei allen XML-Daten auf, die Knoten namens »item« enthalten. Überspringen Sie in diesem Fall einfach den Knoten, der den Fehler ausgelöst hat, indem Sie manuell den nächsten Knotennamen angeben, also »item«:

```
PS > $xml.rss.channel.item
```

```
title      : Dioxin-Schweine: Verdächtiges Fleisch landete auch in
              Polen und Tschechien
link       : http://www.spiegel.de/wirtschaft/service/0,1518,73952
              8,00.html#ref=rss
description : Möglicherweise mit Dioxin belastetes Fleisch von Schw
              einen ging in den Handel und wurde auch nach Polen un
              d Tschechien verkauft. Laut Verbraucherministerium ka
              nn keine Ware mehr zurückgeholt werden. Bei Tests von
              Eiern ergab mehr als ein Viertel der Proben einen zu
              hohen Dioxin-Gehalt.
category   : Wirtschaft
pubDate    : Fri, 14 Jan 2011 12:56:32 +0100
guid       : http://www.spiegel.de/wirtschaft/service/0,1518,73952
              8,00.html
encoded    : encoded
enclosure  : enclosure

title      : Vatikan: Ex-Papst Johannes Paul II. wird seliggesproc
(...)

```

Mit *Select-Object* bestimmen Sie aber selbst, welche Eigenschaften ausgegeben werden sollen und welche nicht:

```
PS > $xml.rss.channel.item | Format-Table Title, Link, Category
```

title	link	category
-----	----	-----
Dioxin-Schweine: Verdäc...	http://www.spiegel.de/w...	Wirtschaft
Vatikan: Ex-Papst Johan...	http://www.spiegel.de/p...	Panorama
Achilles' Ferse: Kaval...	http://www.spiegel.de/s...	Sport
Mercedes Benz 219, Bauj...	http://www.spiegel.de/a...	Auto
Fall "Ruby": Justiz erm...	http://www.spiegel.de/p...	Politik
Aktivurlaub in Pjöngjan...	http://www.spiegel.de/r...	Reise
Vorratsdatenspeicherung...	http://www.spiegel.de/n...	Netzwelt
Massendemo gegen Ben Al...	http://www.spiegel.de/p...	Politik
Trendwende: Deutsche ca...	http://www.spiegel.de/r...	Reise
Mordprozess: Polizistin...	http://www.spiegel.de/p...	Panorama
(...)		

Sie können so auch bestimmen, wie viele *items*-Elemente es gibt:

```
PS > ($xml.rss.channel.item).Count
20
```

Da gleichnamige Elemente eines Feeds (sogenannte »Zwillinge« oder »Siblings«, generell also alle mehrfach vorkommenden Knoten einer XML-Datei in demselben Abschnitt) als Feld (Array) gespeichert werden, ist der Zugriff auf einzelne Elemente möglich, indem Sie den gewünschten Index in eckigen Klammern angeben. Die nächste Zeile liefert den fünften Beitrag des Feeds (da der Index mit 0 beginnt, lautet dessen Indexzahl entsprechend 4):

```
PS > $xml.rss.channel.item[4]
```

Der Inhalt einer geladenen XML-Datei kann mit *Save()* auch wieder als XML-Datei gespeichert werden:

```
PS > $xml.Save("$home\test.xml")
```

Eine sehr wichtige Methode dieses Objektmodells heißt *SelectNodes()*. Mit ihr können sogenannte *XPath*-Abfragen ausgeführt werden, zu denen Sie im folgenden Abschnitt mehr erfahren:

```
PS > $infos.SelectNodes("//item")
```

Möchten Sie an die ausgeblendeten Eigenschaften und Methoden herankommen, verwenden Sie jeweils *PSBase*:

```
PS > $infos.rss.PSBase.Attributes
#text
-----
2.0
http://purl.org/rss/1.0/modules/content/
```

Mit XPath Informationen in XML finden

Sie möchten aus einer komplexen XML-Datei ganz bestimmte Informationen heraussuchen oder auf bestimmte Elemente/Knoten der XML-Datei gezielt zugreifen.

Lösung

Verwenden Sie die XML-Abfragesprache *XPath*, indem Sie Ihre *XPath*-Abfrage mit der Methode *SelectNodes()* stellen. Die folgende Abfrage liefert alle *item*-Elemente aus einem RSS-Feed:

```
PS > $xml = New-Object XML
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
PS > $xml.SelectNodes("//item")
```

Diese Abfragesprache enthält zahlreiche Funktionen wie zum Beispiel *string-length()*, mit deren Hilfe sich auch ausgefallenerere Abfragewünsche realisieren lassen. Die folgende Abfrage liefert alle *item*-Elemente, deren Titel kürzer als 40 Zeichen ist:

```
PS > $xml.SelectNodes("//item[string-length(title)<40]") | select Title, Link
title                                     link
-----
Russland - Putin neuer Regierungschef   http://www.focus.de/politik/ausland/
russlan...
Verbraucher - Angelockt und abgezockt    http://www.focus.de/finanzen/recht/tid-
6551...
Lebensmittel - Absage an die Ampel       http://www.focus.de/gesundheit/ernaehrung/
n...

PS > $xml.SelectNodes("//item[string-length(title)<40]/title")
#text
-----
Russland - Putin neuer Regierungschef
Verbraucher - Angelockt und abgezockt
Lebensmittel - Absage an die Ampel
```

ACHTUNG *XPath* unterscheidet zwischen Groß- und Kleinschreibung. Achten Sie also darauf, die Namen der Elemente in der richtigen Schreibweise anzugeben.

Hintergrund

XPath ist eine Abfragesprache für komplexere XML-Dokumente. Allerdings ist die Abfragesprache *XPath* kryptisch und zunächst relativ schwer zu verstehen. Eine komplette *XPath*-Referenz würde zwar den Rahmen dieses Buchs sprengen, aber die wichtigsten Funktionen finden Sie nachfolgend in Form kurzer Beispiele. Sie verdeutlichen kurz und prägnant, wie *XPath* funktioniert.

HINWEIS

Denken Sie daran: *XPath* unterscheidet zwischen Groß- und Kleinschreibung! In RSS-Feeddateien lautet der Name jedes Schlagzeileintrags »item«. Geben Sie stattdessen »Item« an, wird nichts gefunden.

Der einfache Schrägstrich kennzeichnet absolute Pfade. Die folgende Zeile würde also alle *item*-Elemente in *rss/channel* auflisten:

```
PS > $xml = New-Object XML
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
PS > $xml.SelectNodes("/rss/channel/item")
```

Demgegenüber wirkt sich der doppelte Schrägstrich relativ aus und findet sämtliche Elemente dieses Namens irgendwo im Dokument. Die folgende Zeile findet alle *item*-Elemente unabhängig von ihrem genauen Speicherort:

```
PS > $xml.SelectNodes("//item")
```

Der Stern ist ein Platzhalterzeichen für sämtliche folgenden Pfade. Die folgende Zeile würde deshalb alle Elemente auflisten, die sich irgendwo im Zweig *rss/channel* befinden:

```
PS > $xml.SelectNodes("/rss/channel/*")
```

Ähnlich funktioniert das nächste Beispiel, das alle *item*-Elemente auflistet, die sich zwei Ebenen unterhalb der Wurzel befinden:

```
PS > $xml.SelectNodes("/*/*/item")
```

Entsprechend listet das folgende Beispiel sämtliche Elemente im XML-Dokument auf:

```
PS > $xml.SelectNodes("/*/*")
```

Eckige Klammern sprechen einzelne Elemente in einer Aufzählung an. Den fünften *item* erhalten Sie also zum Beispiel so:

```
PS > $xml.SelectNodes("//item[4]")
```

Anstelle einer festen Zahl sind auch Funktionen wie *last()* erlaubt, die Ihnen das letzte Element liefert:

```
PS > $xml.SelectNodes("//item[last()]")
```

Mit dem Klammeraffen (»@«) sprechen Sie Attribute an. Attribute sind Informationen, die innerhalb eines Elements definiert sind. Die folgende Zeile liefert alle Attribute namens *version*:

```
PS > $xml.SelectNodes("//@version")
```

Alle Elemente, die über ein Attribut namens *version* verfügen, werden so aufgelistet:

```
PS > $xml.SelectNodes("//*[@version]")
```

Mit *not()* drehen Sie die Auswahl um. Möchten Sie alle Elemente sehen, die kein Attribut namens *version* besitzen, gehen Sie so vor:

```
PS > $xml.SelectNodes("//*[@not(@version)]")
```

Sie können auch gezielt nach Attributinhalt suchen. Interessieren Sie sich nur für Elemente, deren *version*-Attribut den Wert »2.0« enthält, ist dies die richtige Formulierung:

```
PS > $xml.SelectNodes("//*[@version='2.0']")
```

Falls Attribute störende Leerzeichen enthalten, können diese vor dem Vergleich mit *normalize-space()* entfernt werden:

```
PS > $xml.SelectNodes("//*[@normalize-space(@version)='2.0']")
```

Elemente werden mit *count()* gezählt. Die folgende Anweisung liefert alle Elemente mit genau einem Child-Element:

```
PS > $xml.SelectNodes("//*[@count(*)=1]")
```

Mit *name()* finden Sie Elemente basierend auf ihrem Namen. Die nächste Anweisung liefert alle Elemente namens *item*:

```
PS > $xml.SelectNodes("//*[@name()='item']")
```

Nützlich ist das in Zusammenhang mit *starts-with()*. Die folgenden Zeilen liefern alle Elemente, bei denen der Name mit dem Buchstaben »i« beginnt:

```
PS > $xml.SelectNodes("//*[@starts-with(name(),'i')]")
```

Verwenden Sie anstelle von *starts-with()* die Funktion *contains()*, wenn Sie die Zeichenfolge irgendwo im Namen suchen. Interessieren Sie sich nur für RSS-*items*, bei denen der Titel das Wort »Birma« enthält, formulieren Sie so:

```
PS > $xml.SelectNodes("//item[contains(title,'Birma')]") | Select Title, Link
title                                     link
-----
Hilfe für Birma - Deutschland stellt... http://www.focus.de/panorama...
Birma - Militärvertreter befürchtet ... http://www.focus.de/panorama...
```

Möchten Sie eine Auswahl auf Basis der Textlänge treffen, setzen Sie *string-length()* ein. Auf diese Weise finden Sie alle *item*-Elemente, deren Titel mindestens 50 Zeichen lang ist:

```
PS > $xml.SelectNodes("//item[string-length(title)>49]") | Select Title, Link
```

Wollen Sie nur *item*-Einträge sehen, deren *category* auf »Politik« lautet, gehen Sie so vor:

```
PS > $xml.SelectNodes("//item[category='Politik']") | Select Title, Link
```

Mit dem Operator »|« führen Sie ein logisches »Oder« aus. Erlaubt sind also beide Ausdrücke. Die folgende Zeile liefert alle *item*-Einträge, die entweder das Wort »Birma« oder »New York« enthalten:

```
PS > $xml.SelectNodes("//item[contains(title,'Birma')] | //item[contains(title,'New York')]") | Select-Object Title, Link
```

Mit *following-sibling::* und *preceding-sibling::* sprechen Sie die folgenden und vorherigen Elemente auf gleicher Ebene an. Die folgende Zeile liefert das vorletzte *item*-Element

```
PS > $xml.SelectNodes("//item[last()]/preceding-sibling::item[1]")
```

Darüber hinaus gibt es mathematische Operatoren wie *mod* oder *div* und mathematische Funktionen wie *ceiling()*. Die folgende Anweisung liefert nur jedes zweite *item*-Element, indem die aktuelle Position des Elements aus *position()* verglichen wird:

```
PS > $xml.SelectNodes("//item[position() mod 2 = 0]")
```

Ebenso könnten Sie sich nur das Element in der Mitte der Aufzählung ausgeben lassen:

```
PS > $xml.SelectNodes("//item[position() = floor(last() div 2 + 0.5) or position() = ceiling(last() div 2 + 0.5)]")
```

Die folgende Anweisung liefert nur die ersten fünf *item*-Elemente:

```
PS > $xml.SelectNodes("//item[position() < 5]")
```

Entsprechend liefert die nächste Anweisung nur die letzten fünf *item*-Elemente:

```
PS > $xml.SelectNodes("//item[position() > (last()-5)]")
```


XML-Informationen filtern

Sie möchten Informationen aus einer XML-Datei mit der PowerShell-Pipeline filtern, um nur die für Sie interessanten Informationen zu sehen.

Lösung

Greifen Sie auf die XML-Datei zu und senden Sie den Inhalt in die PowerShell-Pipeline. Mit *Where-Object* (Kurzform »?*<*«) greifen Sie dann auf die einzelnen Informationen zu und wählen diejenigen aus, die Sie interessieren.

Die folgenden Zeilen laden einen gespeicherten RSS-Feed in ein XML-Objekt. Die Schlagzeilen im RSS-Feed werden in die PowerShell-Pipeline gesendet. *Where-Object* (Kurzform »?*<*«) wählt daraus nur die Schlagzeilen aus, die das Wort »sinken« enthalten:

```
PS > $xml = New-Object xml
PS > $xml.Load('http://www.spiegel.de/schlagzeilen/index.rss')
PS > $xml.rss.channel.item | Where-Object { $_.title.Contains('sinken') }
```

title	link
-----	----
Arbeitskreis Steuerschätzun...	http://www.spiegel.de/wirtschaft/0,151...

Hintergrund

Die Informationen in einem XML-Dokument werden von PowerShell über Eigenschaften direkt zugänglich gemacht. In RSS-Feeds finden sich die Schlagzeilen zum Beispiel immer im Knoten *channel*, der im Knoten *rss* zu finden ist, und die einzelnen Schlagzeilen entsprechen *item*-Elementen. Deshalb liefert die folgende Zeile sämtliche Schlagzeilen:

```
PS > $xml.rss.channel.item
```

Jedes *item*-Element enthält seinerseits Informationen, die wiederum über Eigenschaften zur Verfügung stehen. Diese Eigenschaften hängen ein wenig vom RSS-Feed ab, den Sie ausgewählt haben, aber die Eigenschaften *title* und *link* sind immer vorhanden. Eine Übersicht über alle vorhandenen Eigenschaften liefert die nächste Zeile, indem sie das erste *item*-Element an *Format-List* sendet und *Format-List* mit dem Stern anweist, sämtliche Eigenschaften untereinander auszugeben:

```
PS > $xml.rss.channel.item[0] | Format-List *
title : Spritpreise: Diesel klettert auf neuen Rekordstand
link  : http://www.spiegel.de/auto/aktuell/0,1518,552246,00.html
```

Weil diese Informationen über Eigenschaften ansprechbar sind, können Sie die Ausgabe sehr bequem filtern. Dazu leiten Sie die *item*-Elemente via Pipeline in *Where-Object* (Kurzform »?*<*«).

Darin können Sie dann beliebige Bedingungen formulieren, die erfüllt sein müssen, damit das *item*-Objekt durch die Pipeline gelassen wird.

Die folgende Zeile macht sich die Eigenschaften *title* und *link* zunutze und liefert nur *item*-Elemente, deren Titel mindestens 30 Zeichen lang ist und mit dem Wort »Bundesliga« beginnen. Der Link muss das Wort »fussball« enthalten:

```
PS > $xml.rss.channel.item | ? { $_.Title.Length -gt 30 -and $_.Title.StartsWith('Bundesliga') -and $_.Link.Contains('fussball') }
```

title	link
Bundesliga-Kommentar: Frankfu...	http://www.spiegel.de/sport/fussbal...

ACHTUNG Denken Sie an Groß- und Kleinschreibung! *\$_Link.Contains()* unterscheidet zwischen Groß- und Kleinbuchstaben. Wollen Sie das nicht, könnten Sie Texte vor dem Vergleich zuerst mit *toLower()* in Kleinbuchstaben umwandeln:

```
PS > $xml.rss.channel.item | ? { $_.Title.Length -gt 30 -and $_.Title.StartsWith('Bundesliga') -and $_.Link.ToLower().Contains('FUSSbal1'.ToLower()) }
```

title	link
Bundesliga-Kommentar: Frankfu...	http://www.spiegel.de/sport/fussbal...

Natürlich funktionieren die Beispiele bei Ihnen gegebenenfalls anders, abhängig von den XML-Daten, die Sie verwenden. Passen Sie die Filterkriterien also entsprechend an.

Neues XML-Dokument erstellen

Sie möchten eigene Daten in Form eines XML-Dokuments speichern.

Lösung

Der einfachste Weg ist das Cmdlet *Export-Clixml*, mit dem Sie Ergebnisse in einem standardisierten XML-Format speichern und später mit *Import-Clixml* wieder einlesen können:

```
PS > Get-Process | Export-Clixml ausgabe.xml
PS > $liste = Import-Clixml ausgabe.xml
```

Alternativ können Sie XML-Dokumente aber auch manuell aus den dafür notwendigen Elementen mithilfe der Methoden des XML-Objektmodells zusammensetzen:

```

PS > $info = New-Object xml
PS > $root = $info.CreateElement('Personen')
PS > $el = $info.CreateElement('Person')
PS > $el.set_InnerText('Tobias')
PS > $att = $info.CreateAttribute('male')
PS > $att.PSBase.Value = 'true'
PS > $el.SetAttributeNode($att)
#text
----
true
PS > $root.AppendChild($el)
male                                     #text
----                                     ----
true                                     Tobias
PS > $info.AppendChild($root)

Person
-----
Person

PS > $info.PSBase.InnerXml
<?xml version="1.0" encoding="iso-8859-1"?><Personen><Person male="true">Tobias</Person></
Personen>

PS > $info.Save("$home\info.xml")
PS > . $home\info.xml
PS > $info.Personen.Person
PS > $info.Personen.Person

male                                     #text
----                                     ----
true                                     Tobias

```

Hintergrund

Die Cmdlets *Export-Clixml* und *Import-Clixml* machen es sehr einfach, Objekte und Ergebnisse in XML-Form zu serialisieren. Die Cmdlets kümmern sich dabei automatisch darum, die Eigenschaften der Objekte zu beschreiben und in gültigem XML als Datei zu speichern beziehungsweise aus einer Datei zu lesen.

Aufwändiger, aber auch flexibler ist es, XML-Dateien komplett selbst zu konstruieren. Dazu dienen die verschiedenen Methoden des XML-Objektmodells. Hierbei kommt es allerdings darauf an, dass Sie selbst bei der Konstruktion des XML die notwendigen Konventionen einhalten, damit gültiges XML entsteht.

Neues XML-Dokument aus Schablone erstellen

Sie möchten auf möglichst einfachem Weg Rohdaten in einem XML-Format speichern.

Lösung

Legen Sie sich eine XML-Schablone an, die die unumgänglichen Basiselemente eines XML-Dokuments definiert. Die eigentlichen Daten, die im XML-Dokument verpackt werden sollen, werden über einen Unterausdruck eingelesen, zum Beispiel aus einer kommaseparierten Liste.

Legen Sie zuerst eine kommaseparierte Liste an, zum Beispiel eine Aufstellung von Benutzern. Diese Liste kann aus einer beliebigen Quelle stammen:

```
PS > $benutzer = @'
>> Vorname,Nachname,Gruppe
>> Tobias,Weltner,Administrator
>> Martina,Weltner,Benutzer
>> Cofi,Zumsewitz,Gast
>> Anna,Litkowski,Benutzer
>> '@
>>
PS > $benutzer | Set-Content $home\benutzerliste.csv
PS > Import-Csv $home\benutzerliste.csv
```

Vorname	Nachname	Gruppe
-----	-----	-----
Tobias	Weltner	Administrator
Martina	Weltner	Benutzer
Cofi	Zumsewitz	Gast
Anna	Litkowski	Benutzer

Als Nächstes definieren Sie das XML-Grundgerüst mit einem *Here-String*, der doppelte Anführungszeichen verwendet. So werden Unterausdrücke ausgeführt. Innerhalb eines solchen Unterausdrucks wird der Inhalt der Datei mit den kommaseparierten Daten in die XML-Struktur eingebunden:

```

PS > $xml = @"
<?xml version="1.0" encoding="UTF-8" ?>
<konten>
<description>Liste der Benutzerkonten</description>
$(Import-Csv $home\benutzerliste.csv| Foreach {
'<person>'
"<vorname>${_.Vorname}</vorname>"
"<nachname>${_.Nachname}</nachname>"
"<gruppe>${_.Gruppe}</gruppe>"
'</person>'
})
</konten>
"@
PS > $xml
<?xml version="1.0" encoding="UTF-8" ?><konten><description>Liste der Benutzerkonten</
description><person> <vorname>Tobias</vorname> <nachname>Weltner</nachname>
<gruppe>Administrator</gruppe> </person> <person> <vorname>Martina</vorname>
<nachname>Weltner</nachname> <gruppe>Benutzer</gruppe> </person> <person> <vorname>Cofi</
vorname> <nachname>Zumsewitz</nachname> <gruppe>Gast</gruppe> </person> <person>
<vorname>Anna</vorname> <nachname>Litkowski</nachname> <gruppe>Benutzer</gruppe> </
person></konten>

```

Nun wird der XML-Rohtext in das XML-Format konvertiert und kann als XML-Datei eingesetzt werden:

```

PS > $xml = [xml]$xml
PS > $xml.konten

description                person
-----
Liste der Benutzerkonten   {Tobias, Martina, Cofi, Anna}

PS > $xml.konten.person
vorname      nachname      gruppe
-----
Tobias       Weltner       Administrator
Martina      Weltner       Benutzer
Cofi         Zumsewitz     Gast
Anna         Litkowski     Benutzer

PS > $xml.konten.person | ? { $_.gruppe -like 'Benutzer' }
vorname      nachname      gruppe
-----
Martina      Weltner       Benutzer
Anna         Litkowski     Benutzer

```

Hintergrund

Oft ist es sehr viel einfacher, eine XML-Datei als Rohtext anzugeben, als die XML-Struktur über das XML-Objektmodell zu erstellen. Allerdings funktioniert dies nur, wenn Sie streng auf die korrekte XML-Strukturierung achten und sogenannten »wohlgeformten« XML-Code erstellen.

Weil es häufig nur darum geht, Rohdaten in einer XML-Struktur zu verpacken, kann man sich den Erstellungsprozess erleichtern, indem man ihn automatisiert. Ein sehr einfacher Weg ist dabei, die XML-Grundstruktur als Text bereitzustellen und in diese Vorlage die eigentlichen Daten einzufügen.

Der effizienteste Weg hierfür sind *Here-Strings*, die mit »@"« eingeleitet und mit »"@« beendet werden. Dazwischen kann beliebiger Text angegeben werden. Die doppelten Anführungszeichen sorgen dafür, dass Variablen und Unterausdrücke durch ihre jeweiligen Werte ersetzt werden, und genau das macht sich das Beispiel zunutze. Es liest die Rohdaten mit *Import-Csv* aus einer kommaseparierten Liste ein und fügt diese dann dynamisch mit einer *Foreach*-Schleife in den Text ein.

Das Ergebnis ist beliebig umfangreiches, strukturiertes und wohlgeformtes XML, das nun nur noch mittels *[xml]* in ein XML-Objekt konvertiert zu werden braucht. Danach stehen Ihnen alle Möglichkeiten des XML-Objektmodells zur Verfügung, und Sie können die in der XML-Datei strukturiert vorliegenden Daten nach beliebigen Kriterien filtern und ausgeben.

Im vorliegenden Beispiel werden die Rohdaten aus einer bereits vorhandenen kommaseparierten Liste entnommen. Tatsächlich könnten Sie aber auf ähnlich einfache Weise auch komplexe Objekte automatisiert als XML verpacken. Dazu definieren Sie sich lediglich eine passende Funktion namens *Export-XML*:

```
Function Export-XML($RootTag="WURZEL",$ItemTag="ELEMENT", -->
    $Childs="*", $Attributes=$Null) {
    Begin {
        $xml = "<$RootTag>`n"
    }

    Process {
        $xml += " <$ItemTag"
        if ($Attributes)
        {
            Foreach ($attr in $_ | Get-Member -type *Property $Attributes) {
                $name = $attr.Name
                $xml += " $Name="`"$($_.$Name)`""
            }
        }
        $xml += ">`n"
        Foreach ($child in $_ | Get-Member -Type *Property $Childs)
        {
            $Name = $child.Name
            $xml += "    <$Name>$($_.$Name)</$Name>`n"
        }
        $xml += " </$ItemTag>`n"
    }
}
```

```
End {
    $xml += "</$RootTag>`n"
    [xml]$xml
}
```

HINWEIS

PowerShell selbst bringt bereits ein Cmdlet namens *Export-Clixml* mit, das ebenfalls Objekte in XML-Dateien verpackt. Es dient allerdings vornehmlich dazu, Objekte in einem einheitlichen Format zu serialisieren, damit sie später von *Import-Clixml* wieder eingelesen werden können. Sie haben deshalb bei *Export-Clixml* keinen Einfluss auf die Namen der XML-Elemente und können auch nicht bestimmen, welche Objekteigenschaften als Attribut und welche als Child-Element serialisiert werden. Diese Möglichkeiten bietet Ihnen jedoch nun Ihre eigene Funktion *Export-XML*.

Sie könnten nun beliebige Objekte als XML verpacken, zum Beispiel so:

```
PS > $liste = dir | Export-XML
PS > $liste
<WURZEL>
<ELEMENT>
  <PSChildName>Application Data</PSChildName>
  <PSDrive>C</PSDrive>
  <PSIsContainer>True</PSIsContainer>
  <PSParentPath>Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias</PSParentPath>
  <PSPath>Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias\Application Data</PSPath>
  <PSProvider>Microsoft.PowerShell.Core\FileSystem</PSProvider>
  <Attributes>Directory</Attributes>
  <CreationTime>11/19/2007 21:57:06</CreationTime>
  <CreationTimeUtc>11/19/2007 20:57:06</CreationTimeUtc>
  <Exists>True</Exists>
  <Extension></Extension>
  <FullName>C:\Users\Tobias\Application Data</FullName>
  <LastAccessTime>11/19/2007 21:57:06</LastAccessTime>
  <LastAccessTimeUtc>11/19/2007 20:57:06</LastAccessTimeUtc>
  <LastWriteTime>11/19/2007 21:57:06</LastWriteTime>
  <LastWriteTimeUtc>11/19/2007 20:57:06</LastWriteTimeUtc>
  <Name>Application Data</Name>
  <Parent>Tobias</Parent>
  <Root>C:\</Root>
  <Mode>d----</Mode>
  <ReparsePoint></ReparsePoint>
</ELEMENT>
<ELEMENT>
  <PSChildName>Bluetooth Software</PSChildName>
  <PSDrive>C</PSDrive>
(...)
```

```

PS > # In echtes XML-Objekt verpacken:
PS > $liste = [xml]$liste
PS > $liste.Wurzel.Element
PSPath           : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias\Application Data
PSDrive          : C
LastAccessTimeUtc : 11/19/2010 20:57:06
CreationTime      : 11/19/2010 21:57:06
PSChildName       : Application Data
LastAccessTime    : 11/19/2010 21:57:06
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\Users\Tobias
ReparsePoint      :
Mode              : d----
Root              : C:\
PSProvider        : Microsoft.PowerShell.Core\FileSystem
LastWriteTimeUtc  : 11/19/2010 20:57:06
Exists            : True
FullName          : C:\Users\Tobias\Application Data
Parent            : Tobias
LastWriteTime     : 11/19/2010 21:57:06
Extension         :
CreationTimeUtc   : 11/19/2010 20:57:06
(...)

```

Sehr viel übersichtlicher wird die XML-Datei, wenn Sie Wurzel und Element durch sinnvollere Namen ersetzen und festlegen, welche Objekteigenschaften als XML-Attribut und welche als eigenständige Child-Elemente verpackt werden:

```

PS > $liste = dir $env:windir | Where-Object { $_.PSIsContainer -eq $false } →
    | Export-XML -root Ordner -attribut Datei -Childs CreationTime, →
        Mode,FullName,Name -Attributes Extension,Length
PS > $liste
<Ordner>
<Datei Extension=".htm" Length="9364">
  <CreationTime>04/24/2008 18:30:37</CreationTime>
  <Mode>-a---</Mode>
  <FullName>C:\Users\Tobias\ausgabe.htm</FullName>
  <Name>ausgabe.htm</Name>
</Datei>
<Datei Extension=".bat" Length="24">
  <CreationTime>03/14/2008 11:05:57</CreationTime>
  <Mode>-a---</Mode>
  <FullName>C:\Users\Tobias\autoexec.bat</FullName>
  <Name>autoexec.bat</Name>
</Datei>
<Datei Extension=".vbs" Length="1355">
  <CreationTime>03/31/2008 16:57:01</CreationTime>
  <Mode>-a---</Mode>
  <FullName>C:\Users\Tobias\batch.vbs</FullName>
  <Name>batch.vbs</Name>
</Datei>
(...)

```



```
PS > $!liste.Ordner.Datei | Select-Object Name, Extension, Length
```

Name	Extension	Length
----	-----	-----
ausgabe.htm	.htm	9364
autoexec.bat	.bat	24
batch.vbs	.vbs	1355
benutzerliste.csv	.csv	123
bibliothek.lib	.lib	494
book.xls	.xls	8066
cmdlet.dll	.dll	8704
cmdlet.vb	.vb	7188
ContosoMitarbeiter.xls	.xls	2764

HINWEIS

Mit *ConvertTo-XML* lassen sich ebenfalls XML-Objekte erstellen. Dabei haben Sie allerdings keinen Einfluss auf die Knotennamen:

```
PS > $xml = Dir $env:windir | ConvertTo-XML
```

```
PS > $xml
```

xml	Objects
---	-----
version="1.0"	Objects

```
PS > $xml.Objects.Object[0].property
```

Name	Type	#text
----	-----	-----
PSPath	System.String	Microsoft.PowerShell.C...
PSParentPath	System.String	Microsoft.PowerShell.C...
PSChildName	System.String	Application Data
PSDrive	System.Management.Autom...	C
PSProvider	System.Management.Autom...	Microsoft.PowerShell.C...
PSIsContainer	System.Boolean	True
BaseName	System.String	Application Data
Mode	System.String	d----
Name	System.String	Application Data
Parent	System.IO.DirectoryInfo	w7-pc9
Exists	System.Boolean	True
Root	System.IO.DirectoryInfo	C:\
FullName	System.String	C:\Users\w7-pc9\Applic...
Extension	System.String	
CreationTime	System.DateTime	30.11.2010 12:54:22
(...)		

Möchten Sie von *ConvertTo-XML* kein XML-Objekt zurückerhalten, sondern XML-Text oder ein XML-Dokument, setzen Sie den Parameter *-as* ein. Mit *-depth* bestimmen Sie, wie tief verschachtelt die Objekteigenschaften serialisiert werden:

```
PS > Get-Process powershell | ConvertTo-XML -as String -depth 2
```

Inhalt einer XML-Datei ändern

Sie möchten Teile einer vorhandenen XML-Datei ändern, um beispielsweise darin Daten zu aktualisieren.

Lösung

Laden Sie die vorhandene XML-Datei zuerst mit *Load()* in ein leeres XML-Dokument. Danach nehmen Sie die Änderungen vor und speichern das Ergebnis wieder mit *Save()*:

```
PS > $info = New-Object xml
PS > $info.Load("$home\info.xml")
PS > $info.Personen.Person.'#text' = 'Anna'
PS > $info.Personen.Person.SetAttribute('male', 'false')
PS > $info.psbase.innerXML
<?xml version="1.0" encoding="iso-8859-1"?><Personen><Person male="false">Anna</Person></Personen>
PS > $info.Save("$home\info.xml")
```

Ebenso lassen sich neue Einträge hinzufügen:

```
PS > $info = New-Object xml
PS > $info.Load("$home\info.xml")
PS > $el = $info.CreateElement('Person')
PS > $el.set_InnerText('Tobias')
PS > $att = $info.CreateAttribute('male')
PS > $att.PSBase.Value = 'true'
PS > $el.SetAttributeNode($att)
#text
----
true

PS > $info.Personen.AppendChild($el)
male                                     #text
----                                     ----
true                                    Tobias

PS > $info.psBase.innerXML
<?xml version="1.0" encoding="iso-8859-1"?><Personen><Person male="false">Anna</Person><Person male="true">Tobias</Person></Personen>

PS > $info.Save("$home\info.xml")
```

Hintergrund

Änderungen an bestehenden XML-Daten sind leicht zu bewerkstelligen, weil Sie lediglich auf die entsprechenden Elemente zugreifen und ihnen neue Werte zuweisen. Dabei kommt es allerdings darauf an, ob sich im XML-Dokument nur ein oder mehrere Elemente befinden. Sind es

mehrere, werden sie als Feld repräsentiert, und deshalb müssen Sie dann mit eckigen Klammern auf das gewünschte Element zugreifen:

```
$info.Personen.Person[0].'#text' = 'Anna'
```

Die besondere Eigenschaft *#text* bezeichnet den Inhalt eines Elements. Weil der Name dieser Eigenschaft ein Sonderzeichen enthält, muss die Eigenschaft in Anführungszeichen gestellt werden.

Ein häufig besserer Weg ist der Einsatz von *XPath*. Möchten Sie zum Beispiel den Namen des Eintrags »Anna« ändern, wählen Sie diesen zuerst mit *XPath* aus. Verwenden Sie dazu die Funktion *text()*, die den Inhalt eines XML-Elements repräsentiert:

```
PS > $info = New-Object xml
PS > $info.Load("$home\info.xml")
PS > $info.SelectNodes("//Person[text()='Anna']").Item(0).'#text' = 'Martina'
```

Schwieriger ist es, neue Einträge einer bestehenden XML-Datei hinzuzufügen. In diesem Fall müssen die Elemente zuerst mithilfe der Methoden des XML-Objektmodells hergestellt und dann dem XML-Dokument mit *AppendChild()* hinzugefügt werden.

Einfacher geht dies mitunter, indem die neu hinzuzufügenden Knotenelemente als Rohtext angegeben werden. Dazu muss der Rohtext zuerst in XML konvertiert und danach über *Import-Node()* in den passenden Knotentyp umgewandelt werden. So lassen sich aber sehr bequem weitere Personen der XML-Datei hinzufügen:

```
PS > $info = New-Object xml
PS > $info.Load("$home\info.xml")
PS > $neuePerson = [xml]'<Person male="false">Codi</Person>'
PS > $neuerEintrag = $info.ImportNode($neuePerson.Person, $true)
PS > $info.Personen.AppendChild($neuerEintrag)
male                                #text
----                                ----
false                               Codi

PS > $neuePerson = [xml]'<Person male="true">Norbert</Person>'
PS > $neuerEintrag = $info.ImportNode($neuePerson.Person, $true)
PS > $info.Personen.AppendChild($neuerEintrag)
male                                #text
----                                ----
true                                Norbert

PS > $info.PSBASE.InnerXml
<?xml version="1.0" encoding="iso-8859-1"?><Personen><Person male="false">Martina</
Person><Person male="true">Tobias</Person><Person male="false">Codi</Person><Person
male="true">Norbert</Person></ Personen>
```

Alternativ können Sie auch die erweiterten Daten als Roh-XML hinzufügen. Dazu greifen Sie mit *.PSBase.innerXML* auf das Roh-XML des gewünschten Knotens zu und ersetzen oder erweitern den Inhalt:

```
PS > $info = New-Object xml
PS > $info.Load("$home\info.xml")
PS > $info.Personen.PSBase.innerXML
<Person male="true">Tobias</Person>

PS > $info.Personen.PSBase.innerXML += '<Person male="false">Martina</Person>'
PS > $info.Personen.Person
male                #text
----              -----
true                Tobias
false               Martina
```

Zusammenfassung

Anders als in vielen anderen Skriptsprachen muss die komplexe Struktur von XML-Daten in PowerShell nicht aufwändig geparkt und interpretiert werden. Es genügt, sich ein XML-Objekt zu beschaffen und in dieses die XML-Daten aus einer XML-Datei oder einer Internetadresse zu laden.

Das XML-Objekt interpretiert die Struktur der XML-Daten automatisch. PowerShell macht die Knoten der XML-Daten als Untereigenschaften sichtbar. So kann man sich mit der Punkt-schreibweise von Knoten zu Knoten bis zu den jeweils relevanten Daten vortasten.

Alternativ steht auch die XML-Abfragesprache *XPath* zur Verfügung, mit der man beispielsweise bestimmte Knotennamen suchen und ausgeben kann.

Ändern Sie den Inhalt eines XML-Objekts, können die geänderten XML-Daten mit der Methode *Save()* wieder in eine reguläre XML-Datei zurückgeschrieben werden.

Kapitel 15

Mit Datenbanken arbeiten

In diesem Kapitel:

Datenbankunterstützung testen	430
Über .NET Framework auf Datenbanken zugreifen	431
Mit Datasets Ergebnisdaten verarbeiten	441
XML mit SQL aus Microsoft SQL Server abrufen	443
Über COM auf eine Datenbank zugreifen	445
Datenbankinhalte ändern	446
Verfügbare SQL Server-Instanzen ermitteln	448
Zusammenfassung	449

Datenbanken werden in vielen Unternehmen als wichtige Datenquelle eingesetzt und spielen oft auch eine Rolle bei der Inventarisierung von Systemen und Infrastruktur. PowerShell enthält zwar keine eigenen Cmdlets, um auf Datenbanken zuzugreifen, doch über das zugrunde liegende .NET Framework und ADO.NET ist dieser Zugriff möglich. Wie dies geschieht, erfahren Sie in diesem Kapitel.

Datenbankunterstützung testen

Sie möchten wissen, auf welche Datenbankformate Sie zugreifen können.

Lösung

Erfragen Sie von .NET Framework mit *GetFactoryClasses()* die installierten Datenbankprovider, die auf Ihrem Computer zur Verfügung stehen:

```
PS > [System.Data.Common.DbProviderFactories]::GetFactoryClasses()
Name                Description                InvariantName AssemblyQualifiedName
-----
Odbc Data Prov...   .NET Framework ...   System.Dat...   System.Data.Odbc...
OleDb Data Pro...   .NET Framework ...   System.Dat...   System.Data.Oled...
OracleClient D...   .NET Framework ...   System.Dat...   System.Data.Orac...
SqlClient Data...   .NET Framework ...   System.Dat...   System.Data.SqlC...
SQL Server CE ...   .NET Framework ...   Microsoft....   Microsoft.SqlSer...
```

ACHTUNG Einige Datenbanktreiber stehen möglicherweise nur in 32-Bit-Versionen zur Verfügung. Arbeiten Sie auf einem 64-Bit-System, sehen Sie diese Datenbanktreiber dann in einer normalen PowerShell-Konsole nicht. Starten Sie die 32-Bit-PowerShell-Konsole (*Windows PowerShell (x86)*) und testen Sie darin erneut. Sollte der von Ihnen benötigte Datenbanktreiber nur als 32-Bit-Version vorliegen, beschaffen Sie sich entweder ein Update für 64-Bit-Systeme oder führen Sie Datenbankskripts auf 64-Bit-Systemen in der 32-Bit-PowerShell aus. Eine Lösung hierzu finden Sie im folgenden Abschnitt »Über .NET Framework auf Datenbanken zugreifen«.

Hintergrund

Datenprovider erlauben den Zugriff auf ein bestimmtes Datenbankformat. Welche Provider auf Ihrem System installiert sind, verrät die Methode *GetFactoryClasses()*. Bei diesen »Factories« handelt es sich eigentlich um .NET-Typen, die die tatsächlich benötigten Typen für den eigentlichen Datenbankzugriff beschaffen.

Die klassischen Datenbankprovider heißen *ODBC* und *OLE DB*. ODBC steht für *Open Database Connectivity* und ist ein Standard, mit dem Programmierer und Skriptautoren unkompliziert auf unterschiedlichste Datenbanktypen zugreifen können. Nötig ist dafür jeweils nur der für die Datenbank passende ODBC-Treiber. Seit Windows 2000 ist ODBC als Teil der *MDAC*-Komponente integraler Bestandteil von Windows.

OLE DB (*Object Linking and Embedding for Databases*) ist die Folgeentwicklung von ODBC. Auch OLE DB erlaubt Programmierern und Skriptautoren, unkompliziert auf Datenbanken zuzugreifen, und beide Technologien unterscheiden sich aus Programmiersicht nur durch den verwendeten Connection-String, mit dem der Datenbanktyp und der Ort der Datenbank festgelegt wird. Allerdings ist OLE DB oft schneller und leistungsfähiger als das ältere ODBC.

Neben ODBC und OLE DB gibt es weitere spezifische Datenbankprovider. Windows liefert zum Beispiel die Provider für Microsoft SQL Server mit. Solche nativen Provider stehen aber auch von vielen anderen Datenbankherstellern kostenlos zum Nachrüsten zur Verfügung.

Einige Datenbanktreiber liegen möglicherweise noch in älteren 32-Bit-Versionen vor. Diese funktionieren in 64-Bit-Umgebungen nicht. Wenn Sie beim Versuch, eine Datenbank anzusprechen, auf 64-Bit-Systemen ungewöhnliche Fehlermeldungen erhalten, kann dies die Ursache sein. Sollten Sie nicht in der Lage sein, ein Treiberupdate für 64-Bit-Systeme für den Datenbanktreiber zu beziehen, können Sie PowerShell-Skripts auf 64-Bit-Systemen auch gezielt in einer 32-Bit-PowerShell ausführen lassen.

Das folgende Skript prüft automatisch, ob es in einer 32-Bit-Umgebung ausgeführt wird. Falls nicht, wird es in einer 32-Bit-PowerShell neu gestartet. So ist sichergestellt, dass 32-Bit-Code stets korrekt ausgeführt wird:

```
if ($env:Processor_Architecture -ne "x86") {  
    Write-Warning 'Launching x86 PowerShell'  
    & "$env:windir\syswow64\windowspowershell\v1.0\powershell.exe" →  
        -noninteractive -noprofile -file $myinvocation.Mycommand.path →  
        -executionpolicy bypass  
    exit  
}  
  
"An diesem Punkt wird der Code IMMER in einer 32-Bit-Umgebung →  
ausgeführt, auch auf 64-Bit-Systemen."  
$env:Processor_Architecture  
[IntPtr]::Size
```

Über .NET Framework auf Datenbanken zugreifen

Sie möchten die Methoden von .NET Framework verwenden, um auf eine Datenbank zuzugreifen.

Lösung

Bestimmen Sie zunächst, über welchen Provider Sie die Datenbank ansprechen wollen. Entscheiden Sie sich also für ODBC, OLE DB oder einen spezifischen Provider und wählen Sie den passenden Connection-String für den gewünschten Datenbanktyp aus.

Die folgenden Zeilen lesen den Inhalt der Tabelle *Kunden* aus der Access-Datenbank *c:\skripts\nordwind.mdb* mithilfe des ODBC-Providers (Tabelle 15.1) aus. Dabei erledigt die Funktion *Get-AccessDatabase* den gesamten logischen Teil und liefert die gelesenen Daten in

Form von Objekten an Sie zurück. Diese Objekte können anschließend wie gewohnt in der PowerShell-Pipeline weiterverarbeitet werden:

```
function Get-AccessDatabase {
    param(
        $dbpath = "c:\skripts\nordwind.mdb",
        $sql = "select * from Kunden"
    )

    $providertype = 'System.Data.Odbc' # Groß- und Kleinschreibung
                                     # WICHTIG!
    $connection = "Driver={Microsoft Access Driver (*.mdb)};Dbq=$dbpath"

    $provider = [System.Data.Common.DBProviderFactories] →
                ::GetFactory($providertype)
    $db = $provider.CreateConnection()
    $db.ConnectionString = $connection
    $db.Open()
    $cmd = $provider.CreateCommand()
    $cmd.CommandText = $sql
    $cmd.Connection = $db
    $reader = $cmd.ExecuteReader()
    while ($reader.Read()) {
        $hash = @{}
        for ($x = 0; $x -lt $reader.FieldCount; $x++) {
            $hash.$( $reader.GetName($x)) = $reader.GetValue($x)
        }
        New-Object PSObject -property $hash
    }
    $reader.Close()
    $db.Close()
}
```

PS > **Get-AccessDatabase**

```
Telefon      : 030-0074321
Straße       : Obere Str. 57
Kunden-Code  : ALFKI
Ort          : Berlin
Kontaktperson : Maria Anders
Firma        : Alfreds Futterkiste
Region       :
Land         : Deutschland
PLZ          : 12209
Telefax      : 030-0076545
Position     : Vertriebsmitarbeiterin
```

```
Telefon      : (5) 555-4729
Straße       : Avda. de la Constitución 2222
Kunden-Code  : ANATR
Ort          : México D.F.
```



```

Kontaktperson : Ana Trujillo
Firma        : Ana Trujillo Emparedados y helados
Region       :
Land         : Mexiko
PLZ          : 05021
Telefax      : (5) 555-3745
Position     : InhaberIn
(...)

```

Da Sie der Funktion eine beliebige SQL-Datenbankabfrage übergeben können, können die Daten meist schon serverseitig gefiltert und sortiert werden, sodass die (langsamere) clientseitige Filterung mittels der PowerShell-Pipeline nicht unbedingt notwendig ist:

```

PS > Get-AccessDatabase -sql 'SELECT Firma, Telefon, Land FROM Kunden →
ORDER BY Land DESC'

```

Land	Firma	Telefon
----	-----	-----
Venezuela	LINO-Delicateses	(8) 34-56-12
Venezuela	GROSELLA-Restaurante	(2) 283-2951
Venezuela	HILARIÓN-Abastos	(5) 555-1340
Venezuela	LILA-Supermercado	(9) 331-6954
USA	Let's Stop N Shop	(415) 555-5938
USA	Save-a-lot Markets	(208) 555-8097
USA	Rattlesnake Canyon Grocery	(505) 555-5939
(...)		

Hintergrund

Möchten Sie eine Datenbank über ODBC ansprechen, verwenden Sie je nach Datenbanktyp einen der Connection-Strings aus Tabelle 15.1 und passen die Angaben darin entsprechend an. Geben Sie in Ihren Skripts als *\$providerType* an: *System.Data.Odbc*.

ACHTUNG Geben Sie den Namen des Providers in genau der angegebenen Schreibweise an, denn hier wird ausnahmsweise zwischen Groß- und Kleinschreibung unterschieden!

ODBC-Connection-Strings enthalten immer die Anweisung *Driver=*, hinter der der gewünschte ODBC-Datenbanktreiber genannt wird. Bei serverbasierten Datenbanken geben Sie den Ort der Datenbank dann über *Server=* und *Database=* an. Dateibasierte Datenbanken benennen den Pfad zur Datenbankdatei über *Dbq=*. Darüber hinaus sind weitere Angaben wie zum Beispiel *Uid=* und *Pwd=* möglich, mit denen Sie sich an der Datenbank anmelden oder weitere Spezialeinstellungen vornehmen.

ODBC ist der älteste Providerstandard und bietet deshalb eine gute Abwärtskompatibilität. Die folgende Tabelle gibt eine Übersicht gebräuchlicher ODBC-Connection-Strings. Datenbanken, die in der Tabelle mit einem Stern gekennzeichnet sind, benötigen spezielle ODBC-Treiber, die Sie zunächst vom Hersteller oder aus dem Internet herunterladen und installieren müssen.

Datenbank	ODBC-Connection-String
Access Exklusiver Zugriff	<i>Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\..\datenbank.mdb;Exclusive=1;Uid=admin;Pwd=;</i>
Access Standardsicherheit	<i>Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\..\datenbank.mdb;Uid=Admin;Pwd=;</i>
Active Directory	<i>Provider=ADSDSOObject;</i>
Active Directory mit Anmeldedaten	<i>Provider=ADSDSOObject;User Id=Benutzername;Password=Kennwort;</i>
Cache (*)	<i>DRIVER={InterSystems ODBC};SERVER=ServerAdresse;DATABASE=Datenbankname;UID=Benutzername;PWD=Kennwort;</i>
Excel	<i>Driver={Microsoft Excel Driver (*.xls)};DriverId=790;Dbq=C:\Exceldatei.xls;DefaultDir=C:\Ordner;</i>
Lotus Notes (*)	<i>Driver={Lotus NotesSQL 3.01 (32-bit) ODBC DRIVER (*.nsf)};Server=ServerAdresse;Database=dbPath\Datenbankname.nsf;Uid=Benutzername;Pwd=Kennwort;</i>
Mimer SQL (*)	<i>Driver={MIMER};Database=Datenbankname;Uid=Benutzername;Pwd=Kennwort;</i>
Oracle alte Version	<i>Driver={Microsoft ODBC Driver for Oracle};ConnectionString=OracleServer.world;Uid=Benutzername;Pwd=Kennwort;</i>
Oracle neue Version	<i>Driver={Microsoft ODBC for Oracle};Server=ServerAdresse;Uid=Benutzername;Pwd=Kennwort;</i>
SQL Server Benutzername/Kennwort erfragen	<i>oConn.Properties("Prompt") = 1 Driver={SQL Native Client};Server=ServerAdresse;Database=Datenbankname;</i>
SQL Server Datenbankdatei dynamisch laden	<i>Driver={SQL Native Client};Server=. \SQLEXPRESS;AttachDbFilename=C:\asd\qwe\Datenbanknamefile.mdf;Database=Datenbankname;Trusted_Connection=Yes;</i>
SQL Server Express lokale Verbindung	<i>Driver={SQL Native Client};Server=. \SQLEXPRESS;AttachDbFilename= DataDirectory Datenbanknamefile.mdf;Database=Datenbankname;Trusted_Connection=Yes;</i>
SQL Server Instanz	<i>Driver={SQL Native Client};Server=ServerName\theInstanceName;Database=Datenbankname;Trusted_Connection=yes;</i>
SQL Server mit MARS (Multiple Active Result Sets)	<i>Driver={SQL Native Client};Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;MARS_Connection=yes;</i>
SQL Server Standardsicherheit	<i>Driver={SQL Native Client};Server=ServerAdresse;Database=Datenbankname;Uid=Benutzername;Pwd=Kennwort;</i>
SQL Server Trusted Connection	<i>Driver={SQL Native Client};Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;</i>

Tabelle 15.1 ODBC-Connection-Strings für ausgewählte Datenbanken

Datenbank	ODBC-Connection-String
SQL Server verschlüsselte Netzwerkverbindung	<i>Driver={SQL Native Client};Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;Encrypt=yes;</i>
SQL Server Benutzername/Kennwort erfragen	<i>oConn.Properties("Prompt") = 1 Driver={SQL Server};Server=ServerAdresse;Database=Datenbankname;</i>
SQL Server Standardsicherheit	<i>Driver={SQL Server};Server=ServerAdresse;Database=Datenbankname;Uid=Benutzername;Pwd=Kennwort;</i>
SQL Server Trusted Connection	<i>Driver={SQL Server};Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=Yes;</i>
Textdatei	<i>Driver={Microsoft Text Driver (*.txt;*.csv)};Dbq=c:\txtFilesFolder\;Extensions=asc, csv, tab, txt;</i>

Tabelle 15.1 ODBC-Connection-Strings für ausgewählte Datenbanken (*Fortsetzung*)

OLE DB ist neuer als ODBC und außerdem häufig deutlich leistungstärker. Falls Ihre Datenbank also über einen OLE DB-Treiber verfügt, sollten Sie besser OLE DB anstelle von ODBC verwenden. Eine Übersicht gebräuchlicher OLE DB-Connection-Strings liefert Tabelle 15.2. Passen Sie die Angaben darin entsprechend an und geben Sie als *\$providerType* in Ihren Skripts *System.Data.OleDb* an.

ACHTUNG Geben Sie den Namen des Providers in genau der angegebenen Schreibweise an, denn hier wird ausnahmsweise zwischen Groß- und Kleinschreibung unterschieden!

OLE DB-Connection-Strings geben den Provider hinter *Provider=* an. Die Datenbank wird als *Data Source=* benannt. Darüber hinaus sind zusätzliche Angaben wie zum Beispiel *User ID=*, *Password=*, *Server=* und *Data Base=* möglich.

Datenbanken, die in der Tabelle mit einem Stern gekennzeichnet sind, benötigen zuerst passende OLE DB-Treiber, die Sie vom Hersteller der Datenbank beziehen können und installieren müssen.

Datenbank	OLE DB-Connection-String
Access 2007 mit Kennwort	<i>Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Ordner\Access2007Datei.accdb;Jet OLEDB:Database Password=DatenbanknamePassword;</i>
Access 2007 Standardsicherheit	<i>Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Ordner\Access2007Datei.accdb;Persist Security Info=False;</i>
Access mit Kennwort	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\..\datenbank.mdb;Jet OLEDB:Database Password=DatenbanknamePassword;</i>
Access Standardsicherheit	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\..\datenbank.mdb;User Id=admin;Password=;</i>

Tabelle 15.2 OLE DB-Connection-Strings für verschiedene Datenbanken

Datenbank	OLE DB-Connection-String
Active Directory	<i>Provider=ADSDSOObject;</i>
Active Directory mit Anmeldedaten	<i>Provider=ADSDSOObject;User Id=Benutzername;Password=Kennwort;</i>
AS/400 (*)	<i>Driver={Client Access ODBC Driver (32-bit)};System=Systemname;Uid=Benutzername;Pwd=Kennwort;</i>
Excel	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\ExcelDatei.xls;Extended Properties="Excel 8.0;HDR=Yes;IMEX=1";</i>
Excel 2007 .xlsb	<i>Provider=Microsoft.ACE.OLEDB.12.0;Data Source=c:\Ordner\Excel2007Binärdatei.xlsb;Extended Properties="Excel 12.0;HDR=YES";</i>
Excel 2007 .xslm	<i>Provider=Microsoft.ACE.OLEDB.12.0;Data Source=c:\Ordner\Excel2007Datei.xslm;Extended Properties="Excel 12.0 Macro;HDR=YES";</i>
Excel 2007 .xlsx	<i>Provider=Microsoft.ACE.OLEDB.12.0;Data Source=c:\Ordner\Excel2007Datei.xlsx;Extended Properties="Excel 12.0 Xml;HDR=YES";</i>
HTML-Tabelle	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=http://www.webseiteMIThtmltable.de/tablepage.html;Extended Properties="HTML Import;HDR=YES;IMEX=1";</i>
IBM DB2 (*)	<i>Provider=DB2OLEDB;Network Transport Library=TCPIP;Network Address=XXX.XXX.XXX.XXX;Initial Catalog=Katalogname;Package Collection=PackageName;Default Schema=Schema;User ID=Benutzername;Password=Kennwort;</i>
Informix (*)	<i>Provider=Iifxoledbc.2;Password=Kennwort;User ID=Benutzername;Data Source=Datenbankname@serverName;Persist Security Info=true;</i>
Ingres (*)	<i>Provider=MSDASQL.1;DRIVER=Ingres;SRVR=xxxxx;DB=xxxxx;Persist Security Info=False;Uid=Benutzername;Pwd=Kennwort;SELECTLOOPS=N;Extended Properties="SERVER=xxxxx;DATABASE=xxxxx;SERVERTYPE=INGRES";</i>
Interbase (*)	<i>provider=sibprovider;location=localhost;data source=c:\Datenbanken\gdbsgdb.gdb;user id=SYSDBA;Password=masterkey;</i>
iSeries (*)	<i>Driver={iSeries Access ODBC Driver};System=Systemname;Uid=Benutzername;Pwd=Kennwort;</i>
MySQL (*)	<i>Provider=MySQLProv;Data Source=Datenbankname;User Id=Benutzername;Password=Kennwort;</i>
Oracle Standard Sicherheit	<i>Provider=msdaora;Data Source=Datenbankname;User Id=Benutzername;Password=Kennwort;</i> Oder Oracle-Provider: <i>Provider=OraOLEDB.Oracle;Data Source=Datenbankname;User Id=Benutzername;Password=Kennwort;</i>

Tabelle 15.2 OLE DB-Connection-Strings für verschiedene Datenbanken (Fortsetzung)

Datenbank	OLE DB-Connection-String
Oracle Trusted Connection	<i>Provider=msdaora;Data Source=Datenbankname;Persist Security Info=False;Integrated Security=Yes;</i> oder Oracle-Provider: <i>Provider=OraOLEDB.Oracle;Data Source=Datenbankname;OSAuthent=1;</i>
Paradox (*)	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\Datenbankname;Extended Properties=Paradox 5.x;</i>
Pervasive (*)	<i>Provider=PervasiveOLEDB;Data Source=C:\path;</i>
Postgre (*)	<i>Provider=PostgreSQL OLE DB Provider;Data Source=ServerAdresse; location=Datenbankname;User ID=Benutzername;password=Kennwort;timeout=1000;</i>
SQL Server Express lokal	<i>Provider=SQLNCLI;Server=. \SQLEXPRESS;AttachDbFilename= DataDirectory DatenbanknameDatei.mdf; Database=Datenbankname;Trusted_Connection=Yes;</i>
SQL Server Benutzername/Kennwort erfragen	<i>oConn.Properties("Prompt") = 1</i> <i>oConn.Open</i> <i>"Provider=SQLNCLI;Server=ServerAdresse;DataBase=Datenbankname;</i>
SQL Server Express Datenbank dynamisch laden	<i>Provider=SQLNCLI;Server=. \SQLEXPRESS;AttachDbFilename=c:\asd\qwe\Datenbanknamefile.mdf; Database=Datenbankname;Trusted_Connection=Yes;</i>
SQL Server Instanz	<i>Provider=SQLNCLI;Server=ServerName\theInstanceName;Database=Datenbankname;Trusted_Connection=yes;</i>
SQL Server mit MARS (Multiple Active Result Sets)	<i>Provider=SQLNCLI;Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;MarsConn=yes;</i>
SQL Server Standardsicherheit	<i>Provider=SQLNCLI;Server=ServerAdresse;Database=Datenbankname;Uid=Benutzername;Pwd=Kennwort;</i>
SQL Server Trusted Connection	<i>Provider=SQLNCLI;Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;</i>
SQL Server Verschlüsselte Netzwerkdaten	<i>Provider=SQLNCLI;Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=yes;Encrypt=yes;</i>
SQL Server Benutzername/Kennwort erfragen	<i>oConn.Provider = "sqloledb"</i> <i>oConn.Properties("Prompt") = 1</i> <i>Data Source=ServerAdresse;Initial Catalog=Datenbankname;</i>
SQL Server Instanz verbinden	<i>Provider=sqloledb;Data Source=ServerName\theInstanceName;Initial Catalog=Datenbankname;Integrated Security=SSPI;</i>
SQL Server mit IP-Adresse	<i>Provider=sqloledb;Data Source=10.10.10.34,1433;Network Library=DBMSSOCN;Initial Catalog=Datenbankname;User ID=Benutzername;Password=Kennwort;</i>
SQL Server Standardsicherheit	<i>Provider=sqloledb;Data Source=ServerAdresse;Initial Catalog=Datenbankname;User Id=Benutzername;Password=Kennwort;</i>

Tabelle 15.2 OLE DB-Connection-Strings für verschiedene Datenbanken (Fortsetzung)

Datenbank	OLE DB-Connection-String
SQL Server Trusted Connection	<i>Provider=sqloledb;Data Source=ServerAdresse;Initial Catalog=Datenbankname;Integrated Security=SSPI;</i>
SQLBase (*)	<i>Provider=SQLBaseOLEDB;Data source=ServerAdresse;Location=Datenbankname;UserId=Benutzername;Password=Kennwort;</i>
Sybase (*)	<i>Provider=ASAProv;Data source=Datenbankname;</i>
Teradata (*)	<i>Provider=TDOLDB;Data Source=ServerAdresse;Persist Security Info=True;User ID=Benutzername;Password=Kennwort;Session Mode=ANSI;</i>
Textdatei feste Spaltenbreite	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\txtFilesFolder\;Extended Properties="text;HDR=Yes;FMT=Fixed";</i>
Textdatei mit Trennzeichen	<i>Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\txtFilesFolder\;Extended Properties="text;HDR=Yes;FMT=Delimited";</i>

Tabelle 15.2 OLE DB-Connection-Strings für verschiedene Datenbanken (Fortsetzung)

Möchten Sie in der Funktion *Get-AccessDatabase* von oben die Abfrage lieber mit dem neueren OLE DB-Provider vornehmen (Tabelle 15.2), ändern Sie lediglich die ersten Zeilen mit den Providerinformationen innerhalb der Funktion und geben stattdessen einen OLE DB-Connection-String an:

```
$providertype = 'System.Data.OleDb'      # Groß- und Kleinschreibung
                                           # ist hier WICHTIG!
$connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"
```

Der übrige Code bleibt unverändert. Allein durch das Austauschen des für den Datenbanktyp passenden Providers können Sie deshalb mit demselben Code die unterschiedlichsten Datenbanken ansprechen. Falls Sie beispielsweise nicht auf eine Access-Datenbank zugreifen wollen, sondern über spezifische Provider direkt auf Microsoft SQL Server (Tabelle 15.3), passen Sie erneut lediglich die ersten Zeilen für die Verbindungsaufnahme an:

```
$providertype = 'System.Data.SqlClient'  # Groß- und Kleinschreibung
                                           # ist hier WICHTIG!
$connection = "server=WIN-V9GK00HXS5N;Initial Catalog=Northwind;User ID=Tobias;Password=topsecret"
```

ACHTUNG Geben Sie den Namen des Providers jeweils in genau der angegebenen Schreibweise an, denn hier wird anders als sonst in PowerShell penibel zwischen Groß- und Kleinschreibung unterschieden.

Verbindungsart	Zugriff
Standardsicherheit	<i>Data Source=ServerAdresse;Initial Catalog=Datenbankname;User Id=Benutzername;Password=Kennwort;</i> Oder: <i>Server=ServerAdresse;Database=Datenbankname;User ID=Benutzername;Password=Kennwort;Trusted_Connection=False;</i>
Trusted Connection	<i>Data Source=ServerAdresse;Initial Catalog=Datenbankname;Integrated Security=SSPI;</i> Oder <i>Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=True;</i>
Instanz	<i>Server=ServerName\theInstanceName;Database=Datenbankname;Trusted_Connection=True;</i>
IP-Adresse	<i>Data Source=190.190.200.100,1433;Network Library=DBMSSOCN;Initial Catalog=Datenbankname;User ID=Benutzername;Password=Kennwort;</i>
Packet Size	<i>Server=ServerAdresse;Database=Datenbankname;User ID=Benutzername;Password=Kennwort;Trusted_Connection=False;Packet Size=4096;</i>
Von CE-Gerät	<i>Data Source=ServerAdresse;Initial Catalog=Datenbankname;Integrated Security=SSPI;User ID=Domäne\Benutzername;Password=Kennwort;</i>
Mit MARS (Multiple Active Result Sets)	<i>Server=ServerAdresse;Database=Datenbankname;Trusted_Connection=True;MultipleActiveResult Sets=true;</i>
Datenbank in SQL Server Express laden	<i>Server=. \SQLExpress;AttachDbFilename=c:\asd\qwe\Datenbanknamefile.mdf;Database=Datenbankname;Trusted_Connection=Yes;</i>
Lokale User-Instanz	<i>Data Source=. \SQLExpress;Integrated Security=true;AttachDbFilename= DataDirectory \Datenbankname.mdf;User Instance=true;</i>

Tabelle 15.3 Zugriff auf Microsoft SQL Server Express über *SqlClient-Provider*

Unabhängig davon, mit welchem Provider Sie sich verbunden haben und auf welche Art von Datenbank Sie zugreifen, läuft der Zugriff nach einheitlichen Schritten ab:

1. Provider-Factory anlegen

Damit der übrige Skriptcode providerunabhängig ist, legen Sie zuerst für den Provider, den Sie nutzen wollen, eine Provider-Factory an. Sie versorgt das Skript später mit allen weiteren providerspezifischen Klassen:

```
$providertype = 'System.Data.OleDb'           # Groß- und Kleinschreibung
                                                # ist hier WICHTIG!
$provider = [System.Data.Common.DBProviderFactories] →
            ::GetFactory($providertype)
```

2. Verbindung zur Datenbank herstellen

Sie erstellen dann einen Connection-String für den Provider, den Sie ausgewählt haben. In diesem Connection-String wird angegeben, auf was für eine Datenbank Sie zugreifen wollen und wo sich diese befindet. Danach beschaffen Sie sich von der Provider-Factory mit *CreateConnection()* ein Verbindungsobjekt und öffnen die Verbindung mithilfe Ihres Connection-Strings:

```
$dbpath = "c:\skripts\nordwind.mdb"
$connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"
$db = $provider.CreateConnection()
$db.ConnectionString = $connection
$db.Open()
```

3. Einen SQL-Befehl erstellen

Nun senden Sie einen SQL-Befehl an die Datenbank, um Daten daraus abzurufen. Hierzu beschaffen Sie sich von der Provider-Factory mit *CreateCommand()* ein *Command*-Objekt und tragen darin die offene Datenbankverbindung und Ihren Befehl ein:

```
$cmd = $provider.CreateCommand()
$sql = "select * from Kunden"
$cmd.CommandText = $sql
$cmd.Connection = $db
```

4. SQL-Befehl ausführen und Ergebnisse auswerten

Danach führen Sie den SQL-Befehl mit *ExecuteReader()* aus. Das Ergebnis ist ein *Reader*-Objekt, mit dem Sie nun die Ergebnisse zeilenweise über *Read()* lesen können. Greifen Sie zum Beispiel mittels *Item()* direkt auf die gewünschten Spalten zu:

```
$reader = $cmd.ExecuteReader()
while ($reader.Read()) {
    $reader.Item('CompanyName')
}
$reader.Close()
```

Kennen Sie die Spaltennamen der Tabelle nicht, lassen sich die Spaltennamen über *GetName()* ermitteln. *GetValue()* liest den Inhalt einer Spalte basierend auf ihrem Index und *FieldCount* liefert die Anzahl der Spalten.

```
$reader = $cmd.ExecuteReader()
while ($reader.Read()) {
    for ($x=0; $x -lt $reader.FieldCount; $x++) {
        '{0,30} = {1,-30}' -f $reader.GetName($x) , $reader.GetValue($x)
    }
    '=' * 40
}
$reader.Close()
```

Das Ergebnis in den letzten beiden Beispielen ist reiner Text und kann deshalb nicht von der PowerShell-Pipeline und den übrigen Cmdlets weiterverarbeitet werden. Mithilfe einer Hash-table verwandeln Sie die Textergebnisse aber sehr einfach in echte Objekte, wobei jede Datenbankspalte zu einer Objekteigenschaft wird:


```
while ($reader.Read()) {  
    $hash = @{}  
    for ($x = 0; $x -lt $reader.FieldCount; $x++) {  
        $hash.$( $reader.GetName($x)) = $reader.GetValue($x)  
    }  
    New-Object PSObject -property $hash  
}
```

Dabei wird zunächst ein leeres Hashtable angelegt (*\$hash = @{}*). Anschließend werden alle Felder des aktuellen Datensatzes ausgelesen und Schlüssel-Wert-Paare in die Hashtable geschrieben. Der Schlüssel entspricht dem Spaltenname des Datensatzes, der Wert dem Inhalt der Spalte. Sind alle Informationen des aktuellen Datensatzes ins Hashtable eingetragen, verwandelt *New-Object* es in ein Objekt. So wird Datensatz für Datensatz in ein echtes Objekt verwandelt.

WICHTIG Vergessen Sie nicht, das *Reader*-Objekt sowie die Datenbankverbindung nach erledigter Arbeit mit *Close()* wieder zu schließen.

Neben *ExecuteReader()* stehen weitere *Execute...*-Befehle zur Verfügung:

- **ExecuteNonQuery()** Führt einen SQL-Befehl aus, der kein Ergebnis zurückliefert, zum Beispiel eine *INSERT INTO*- oder *DELETE*-Anweisung
- **ExecuteScalar()** Liefert nur die erste Spalte der ersten Zeile der Ergebnismenge, zum Beispiel, wenn Sie lediglich die Anzahl von Datensätzen über *COUNT(*)* bestimmen
- **ExecuteRow()** Liefert die erste Zeile des Ergebnisses als *SqlRecord*-Objekt
- **ExecuteXmlReader()** Liefert das Ergebnis als XML zurück

Mit Datasets Ergebnisdaten verarbeiten

Sie möchten die Daten, die Sie von einer Datenbank erhalten, möglichst einfach und komfortabel losgelöst von der Datenbank weiterverarbeiten und zum Beispiel über die PowerShell-Pipeline ausgeben.

Lösung

Speichern Sie die Ergebnisdaten in einem *Dataset*-Objekt. Die Daten werden so von der Datenbank getrennt und können sehr einfach weiterverarbeitet werden. Das folgende Beispiel definiert die Funktion *Get-AccessDatabaseDataset* und setzt die Funktion dann dazu ein, um über die Datenbankabfragesprache SQL ausgewählte Datensätze auszugeben:

```
function Get-AccessDatabaseDataset {  
    param(  
        $dbpath = "c:\skripts\nordwind.mdb",  
        $sql = "select * from Kunden"  
    )  
}
```

```

$dbtype = 'System.Data.OleDb'          # Groß- und Kleinschreibung
                                         # ist hier WICHTIG!
$connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"

$provider = [System.Data.Common.DBProviderFactories] →
             ::GetFactory($dbtype)
$db = $provider.CreateConnection()
$db.ConnectionString = $connection
$db.Open()

$cmd = $provider.CreateCommand()
$cmd.CommandText = $sql
$cmd.Connection = $db

$adapter = $provider.CreateDataAdapter()
$adapter.SelectCommand = $cmd

$dataset = New-Object System.Data.Dataset
$adapter.Fill($dataset, 'Tabelle1') | Out-Null

$db.Close()

$dataset.Tables['Tabelle1']
}

PS > Get-AccessDatabaseDataset -sql 'SELECT [Kunden-Code], Firma, Kontaktperson, →
    [Position] FROM Kunden WHERE [Kunden-Code] LIKE "T%" ORDER BY [Kunden-Code] '

```

Kunden-Code	Firma	Kontaktperson	Position
-----	-----	-----	-----
THEBI	The Big Cheese	Liz Nixon	Marketingm...
THECR	The Cracker Box	Liu Wong	Marketingm...
TOMSP	Toms Spezialitäten	Karin Josephs	Marketingm...
TORTU	Tortuga Restaurante	Miguel Angel Paolino	Inhaber
TRADH	Tradição Hipermercados	Anabela Domingues	Vertriebsm...
TRAIH	Trail's Head Gourme...	Helvetius Nagy	Vertriebsm...

Hintergrund

Aus Geschwindigkeitsgründen kann es sinnvoll sein, die Ergebnisdaten der Datenbank nicht einzeln Datensatz für Datensatz abzurufen, sondern in einem Schritt. Genau dies geschieht hier. Die Ergebnisdaten werden komplett abgerufen und in ein Dataset-Objekt gespeichert. Unmittelbar danach kann die Datenbankverbindung gekappt werden. Für die weitere Arbeit mit den abgerufenen Daten ist diese nicht mehr erforderlich.

Der Einsatz eines Datasets hat weitere Vorteile: Die Datensätze liegen darin bereits automatisch als Objekt vor und müssen nicht erst aufwändig in Objekte verpackt werden. Deshalb ist der Einsatz eines Datasets deutlich schneller.

Allerdings müssen sämtliche Ergebnisse gleichzeitig im Speicher gehalten werden. Wollen Sie mit sehr großen Ergebnismengen arbeiten, sollten Sie deshalb auf Datasets verzichten und die Informationen Datensatz für Datensatz der Reihe nach bearbeiten.

Das Beispiel zeigt außerdem einige Besonderheiten der SQL-Abfragesprache, die zwar nicht im Fokus dieses Buchs steht, aber hier wichtig ist. Möchten Sie auf Spalten zugreifen, deren Namen mit SQL-Schlüsselbegriffen kollidieren oder Sonderzeichen enthalten, stellen Sie die Spaltennamen in eckige Klammern. Sie dürfen das mit allen Spaltennamen tun, aber nötig ist es nur bei solchen, die zu Mißverständnissen führen könnten.

Um ein Stichwort innerhalb einer Spalte zu suchen, verwenden Sie den Operator *LIKE*. Er erwartet das Stichwort in Anführungszeichen. Das Platzhalterzeichen ist bei den meisten SQL-Dialekten nicht wie sonst üblich der Stern («*»), sondern das Prozent-Zeichen («%«).

XML mit SQL aus Microsoft SQL Server abrufen

Sie wollen Daten aus einer Microsoft SQL Server-Datenbank im XML-Format abrufen und weiterverarbeiten.

Lösung

Verwenden Sie in Ihrer SQL-*SELECT*-Anweisung die Anweisung *FOR XML* und geben Sie den Namen des Wurzelements an. Laden Sie dann das Ergebnis der Abfrage in ein neues leeres XML-Objekt. Die folgenden Zeilen lesen die Tabelle *Customers* der Datenbank *Northwind* in ein XML-Objekt:

```
PS > $providertype = 'System.Data.SqlClient'           # Groß- und Kleinschreibung ist
                                                         # hier WICHTIG!
PS > $connection = "server=WIN-V9GK00HXS5N;Initial Catalog=Northwind;
User ID=Tobias;Password=topsecret"
PS >
PS > $sql = "Select * FROM Customers FOR XML AUTO,ROOT('Elements')"
PS >
PS > $provider = [System.Data.Common.DBProviderFactories]::GetFactory($providertype)
PS > $db = $provider.CreateConnection()
PS > $db.ConnectionString = $connection
PS > $db.Open()
PS > $cmd = $provider.CreateCommand()
PS > $cmd.CommandText = $sql
PS > $cmd.Connection = $db
PS > $x = $cmd.ExecuteXMLReader()
PS >
PS > $xml = new-object xml
PS > $xml.load($x)
PS > $db.Close()
PS >
PS > $xml
```

```
Elements
```

```
-----
```

```
Elements
```

```
PS > $xml.Elements
```

```
Customers
```

```
-----
```

```
{Alfreds Futterkiste, Ana Trujillo Emparedados y helados, Antonio Moreno Taquería, Around the Horn...}
```

```
PS > $xml.Elements.Customers
```

```
Address      : Obere Str. 57
Country      : Germany
PostalCode   : 12209
Phone        : 030-0074321
City         : Berlin
CustomerID   : ALFKI
CompanyName  : Alfreds Futterkiste
ContactTitle : Sales Representative
ContactName  : Maria Anders
Fax          : 030-0076545
```

```
(...)
```

```
PS > $xml.SelectNodes("//Customers[@Country='Germany']")
```

```
Address      : Obere Str. 57
Country      : Germany
PostalCode   : 12209
Phone        : 030-0074321
City         : Berlin
CustomerID   : ALFKI
CompanyName  : Alfreds Futterkiste
ContactTitle : Sales Representative
ContactName  : Maria Anders
Fax          : 030-0076545

Address      : Forsterstr. 57
Country      : Germany
PostalCode   : 68306
Phone        : 0621-08460
City         : Mannheim
CustomerID   : BLAUS
CompanyName  : Blauer See Delikatessen
ContactTitle : Sales Representative
ContactName  : Hanna Moos
Fax          : 0621-08924
```

Hintergrund

Der Code unterscheidet sich im Anfangsteil nicht von den Skripts zum Abrufen von Daten im Textformat. Wesentlicher Unterschied ist die Anweisung *FOR XML* im SQL-Befehl und der Einsatz von *ExecuteXMLReader()* anstelle von *ExecuteReader()*.

Das Ergebnis von *ExecuteXMLReader()* ist ein Objekt, das die Datenbankergebnisse als XML lesen kann (aber noch nicht gelesen hat). Deshalb muss die Datenbankverbindung noch so lange geöffnet bleiben, bis dieses Objekt mit *Load()* in ein leeres XML-Objekt geladen wird. Dabei importiert das Objekt dann die XML-Ergebnisdaten in das XML-Objekt. Anschließend kann die Datenbankverbindung geschlossen werden. Die Ergebnisdaten befinden sich nun ähnlich losgelöst von der Datenbank wie in einem Dataset, nur liegen sie diesmal im XML-Format vor.

In Kapitel 14 finden Sie viele weitere Beispiele, die Ihnen zeigen, wie Sie mit *SelectNodes()* und *XPath* die XML-Daten weiterverarbeiten und filtern können.

Über COM auf eine Datenbank zugreifen

Sie möchten möglichst simpel über ODBC oder OLE DB auf eine Datenbank zugreifen oder ältere Datenbankskripts mit möglichst wenig Änderungen nach PowerShell konvertieren.

Lösung

Legen Sie mit *New-Object* das COM-Object *ADODB.Connection* an und verwenden Sie zur Verbindungserstellung wahlweise einen ODBC- oder einen OLE DB-Connection-String. Der folgende Code definiert die Funktion *Get-AccessDatabaseCOM* und greift darüber auf eine Access-Datenbank zu:

```
function Get-AccessDatabaseCOM {
    param(
        $dbpath = "c:\skripts\nordwind.mdb",
        $sql = "select * from Kunden"
    )

    $connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"
    $db = New-Object -comObject ADODB.Connection
    $db.Open($connection)
    $rs = $db.Execute($sql)
    while (!$rs.EOF) {
        $hash = @{}
        foreach ($field in $rs.Fields) {
            $hash.($field.Name) = $field.Value
        }
        $rs.MoveNext()    # WICHTIG, sonst Endlosschleife!
        New-Object PSObject -property $hash
    }
}
```

```
$rs.Close()
$db.Close()
}
```

```
PS > Get-AccessDatabaseCOM -sql 'SELECT Firma, Land, Telefon FROM Kunden →
ORDER BY Firma DESC'
```

Land	Firma	Telefon
----	-----	-----
Polen	Wolski Zajazd	(26) 642-7012
Finnland	Wilman Kala	90-224 8858
USA	White Clover Markets	(206) 555-4112
(...)		

Hintergrund

Geht es Ihnen nur darum, schnell und simpel Daten aus einer Datenbank abzurufen, ist das COM-Objektmodell möglicherweise wegen seines simplen Aufbaus besser geeignet als die Methoden von .NET Framework.

- Es ist wesentlich einfacher zu handhaben, denn es unterscheidet nicht zwischen ODBC und OLE DB, sodass keine unterschiedlichen Klassen oder Provider-Factories nötig sind
- Es unterscheidet auch nicht zwischen Verbindung und Befehl, sodass Sie direkt nach dem Öffnen einer Verbindung mit *Execute()* eine SQL-Anweisung ausführen können
- Da ADO allerdings ein anderes Objektmodell verwendet als ADO.NET, greifen Sie hier mit *Fields* auf die einzelnen Felder zu

ACHTUNG Ganz besonders wichtig bei ADO ist die Anweisung *MoveNext()*. Vergessen Sie diese Anweisung, durchläuft Ihre Schleife ständig denselben Ergebnissatz und kommt zu keinem Ende.

Datenbankinhalte ändern

Sie möchten möglichst einfach auf eine Datenbank zugreifen und darin Daten ändern.

Lösung

Verwenden Sie das Cmdlet *New-Object* mit dem Parameter *-comObject* und legen Sie ein neues *ADODB.Recordset*-Objekt an. Dieses Objekt kann mit den Connection-Strings aus Tabelle 15.1 (Seite 434) und Tabelle 15.2 (Seite 435) sehr einfach eine Verbindung zur gewünschten Datenbank herstellen und dann mit *AddNew()* neue Datensätze hinzufügen:

```
PS > $rs = New-Object -comObject ADODB.Recordset
PS > $dbpath = "c:\skripts\nordwind.mdb"
PS > $connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"
PS > $rs.Open("Kunden",$connection,0,3)
PS > $rs.AddNew()
PS > $rs.Fields["Kunden-Code"].Value = "AB111"
PS > $rs.Fields["Firma"].Value = "TestCompany"
PS > $rs.Fields["Kontaktperson"].Value = "t@btw.com"
PS > $rs.Update()
```

Neue Datensätze dürfen auch direkt per SQL mit *INSERT INTO* eingefügt werden:

```
PS > $sql = 'INSERT INTO Kunden ([Kunden-Code],Firma,Kontaktperson) →
VALUES ("ABC","Scriptinternals","Tobias Weltner")'
PS > $db = New-Object -comObject ADODB.Connection
PS > $dbpath = "c:\skripts\nordwind.mdb"
PS > $connection = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=$dbpath;"
PS > $db.Open($connection)
PS > $db.Execute($sql) | Out-Null
PS > $db.Close()
```

Hintergrund

Änderungen am Inhalt einer Datenbank lassen sich am einfachsten mit den klassischen COM-Objekten realisieren. Zwar kann man alternativ auch die Methoden von .NET Framework einsetzen, doch sind diese ungleich komplexer und resultieren in deutlich mehr Code.

Datensätze lassen sich auf zwei unterschiedliche Arten in eine Datenbank einfügen:

Entweder legen Sie in der gewünschten Tabelle ein neues Recordset an und definieren darin die Inhalte der einzelnen Datensatzspalten. Sobald der gesamte Datensatz beschrieben ist, wird er mit *Update()* in die Datenbank geschrieben. Dieser Ansatz eignet sich besonders dann, wenn Sie sehr viele Informationen in einem Datensatz speichern möchten oder wenn der Datensatz neben Textinformationen auch andere Datentypen wie zum Beispiel Datums- und Zeitinformationen enthält.

Alternativ kann man auch direkt an die Datenbank eine SQL-Anweisung senden, die die Daten einträgt. Dieser Ansatz ist kürzer, aber die Konzeption der dafür notwendigen SQL-Anweisung kann komplex sein. Im Beispiel wird deutlich, dass mißverständliche Spaltennamen (wie zum Beispiel »Kunden-Code«, das ein Sonderzeichen enthält) in eckige Klammern gestellt werden müssen. Noch komplexer kann es werden, wenn Sie einer Spalte ein Datumsformat zuweisen wollen.

Entscheiden Sie sich für den Ansatz mit einem neuen Recordset, müssen beim Öffnen der Datenbank zwei Kennzahlen angegeben werden, die festlegen, was für eine Art von Verbindung Sie wünschen. Die Auswahl legt beispielsweise fest, ob Änderungen an der Datenbank zulässig sind, während Sie darauf zugreifen.

Die erste Kennzahl gibt an, was für eine Art von Datenbankcursor Sie wünschen. Der Cursor bestimmt zum Beispiel, ob Sie sich innerhalb der Datensätze nur vorwärts oder auch rückwärts

bewegen dürfen. Im einfachsten Fall wählen Sie den Wert 0 für *adOpenForwardOnly*. Dieser Cursor beansprucht die wenigsten Systemressourcen.

Konstante	Wert
<i>adOpenForwardOnly</i>	0
<i>adOpenKeyset</i>	1
<i>adOpenDynamic</i>	2
<i>adOpenStatic</i>	3
<i>adOpenUnspecified</i>	-1

Tabelle 15.4 Datenbankcursor-Typen

Der zweite Wert gibt an, wie die Datenbank gesperrt werden soll, wenn Sie Datensätze ändern. Solange die Datenbank gesperrt ist, können andere nicht darauf zugreifen. Im Beispiel wird die Sperrung *adLockOptimistic* mit dem Zahlenwert 3 verwendet. Wählen Sie *adLockReadOnly*, wenn Sie Datenbanken nur lesen, aber nicht ändern möchten.

Konstante	Wert
<i>adLockReadOnly</i>	1
<i>adLockPessimistic</i>	2
<i>adLockOptimistic</i>	3
<i>adLockBatchOptimistic</i>	4
<i>adLockUnspecified</i>	-1

Tabelle 15.5 Sperrungsarten einer Datenbank

WICHTIG Je nach Datenbankdesign gelten für die einzelnen Felder eventuell bestimmte Bedingungen, zum Beispiel erlaubter Datentyp oder die Frage, ob ein Feld leer sein darf oder auf einen bestimmten Wert festgelegt sein muss. Erst wenn Sie *Update()* aufrufen, werden diese Regeln für alle Felder gemeinsam überprüft. Erst hier erhalten Sie bei Unstimmigkeiten also eine Fehlermeldung.

Verfügbare SQL Server-Instanzen ermitteln

Sie möchten wissen, welche SQL-Server auf Ihrem Computer oder innerhalb Ihrer Domäne in Betrieb sind.

Lösung

Verwenden Sie *GetDataSources()*, um die Liste der verfügbaren SQL-Server zu ermitteln:


```
PS > [System.Data.Sql.SqlDataSourceEnumerator]::Instance.GetDataSources()
```

ServerName	InstanceName	IsClustered	Version
-----	-----	-----	-----
WIN-V9GK00HXS5N		No	10.0.1300.13
WIN-V9GK00HXS5N	OFFICESERVERS	No	9.00.3042.00

Hintergrund

Für die Verbindungsaufnahme zu einem SQL-Server wird häufig dessen Name benötigt. Diesen Namen ermittelt *GetDataSources()*.

Zusammenfassung

PowerShell enthält zwar keine Cmdlets, um auf Datenbanken zuzugreifen, doch haben Sie in diesem Kapitel gesehen, dass es relativ einfach möglich ist, die benötigten Funktionen selbst herzustellen.

Der Zugriff auf Datenbanken wird durch ODBC- und OLEDB-Connection-Strings abstrahiert. Sie wählen also lediglich den für Ihren Datenbanktyp geeigneten Connection-String aus. Der übrige Code ist danach portabel, funktioniert also für alle Datenbanktypen gleich. Der Connection-String bestimmt, welcher Datenbanktreiber für den Zugriff eingesetzt wird.

Es kann vorkommen, dass Datenbanktreiber noch nicht 64-Bit-fähig sind. In diesem Fall muss der PowerShell-Code auf 64-Bit-Systemen in der 32-Bit-PowerShell ausgeführt werden. Wie dies automatisiert gewährleistet werden kann, haben Sie im Kapitel erfahren.

Um mit ODBC oder OLEDB arbeiten zu können, werden Befehle benötigt, die entweder .NET Framework oder COM-Komponenten bereitstellen. Beim Lesen von Datenbankinhalten hat .NET Vorzüge, weil es sich nahtlos in das ebenfalls .NET-basierte PowerShell integriert und die Ergebnisse der Datenbankabfrage direkt als Objekte zurückliefern, die danach von anderen Cmdlets in der PowerShell-Pipeline weiterverarbeitet werden können.

Da .NET jedoch insgesamt mehr Code erfordert als COM, ist es beim Einfügen von Informationen oder dem simplen Absenden von SQL-Aktionsanweisungen einfacher, die COM-Objekte zu nutzen.

Kapitel 16

Benutzerverwaltung und Active Directory

In diesem Kapitel:

Active Directory-Cmdlets verwenden	453
Mit Active Directory verbinden	456
Neues Benutzerkonto anlegen	458
Viele neue Benutzerkonten anlegen	460
Benutzerkonto löschen	462
Kennwort eines Benutzerkontos ändern	463
Benutzerkonto suchen	464
Mit LDAP-Filtern suchen	466
Neue Organisationseinheit anlegen	468
Eigenschaften eines Benutzerkontos lesen	469
Eigenschaften eines Benutzerkontos ändern	471
Gruppenmitgliedschaft eines Benutzerkontos auflisten	473
Gruppenmitglieder auslesen	474
Benutzerkonto zu einer Gruppe hinzufügen	474
Benutzerkonto aus einer Gruppe entfernen	475

Auf ein lokales Benutzerkonto zugreifen	476
Auf eine lokale Gruppe zugreifen	477
Lokale Benutzerkonten finden	479
Alle lokalen Gruppen auflisten	480
Prüfen, ob ein lokales Konto existiert	482
Neues lokales Benutzerkonto anlegen	483
Benutzerkonto in eine lokale Gruppe aufnehmen	485
Benutzerkonto aus einer lokalen Gruppe entfernen	486
Lokales Benutzerkonto konfigurieren	487
Prüfen, ob ein lokales Benutzerkonto Mitglied einer Gruppe ist	488
Lokales Benutzerkonto aktivieren oder deaktivieren	489
Lokales Benutzerkonto löschen	490
Zusammenfassung	492

Die Benutzerverwaltung rund um Active Directory bildet einen Kernbereich administrativer Tätigkeit. Während es früher recht schwierig und aufwändig war, Benutzerkonten per Skript anzulegen, zu verwalten oder Reports zu erstellen, ist dies mit PowerShell sehr viel einfacher geworden. Tatsächlich ist die Verwaltung von Active Directory geradezu revolutioniert worden. Jedenfalls dann, wenn für die Verwaltung Cmdlets zur Verfügung stehen. Sie gehören nicht zum Lieferumfang von PowerShell, werden aber von Microsoft kostenlos im Rahmen der RSAT-Tools (Remote Server Administration Tools) ausgeliefert und gehören zum Standardlieferumfang von Windows Server 2008 R2.

Allerdings wurde mit RSAT (und also den Active Directory-Cmdlets von Microsoft) eine neuartige Remotingtechnik eingeführt, die nicht mehr auf dem langsamen und schwerfälligen DCOM basiert, sondern auf Webservices. Sobald Sie mindestens einen Rechner mit Windows Server 2008 R2 als Domänencontroller einsetzen, kann Active Directory mit den Cmdlets verwaltet werden. Ohne einen solchen Server geht dies allerdings auch. Dazu müssen Sie lediglich in Ihrer Infrastruktur einen sogenannten Gatewaydienst installieren, der zwischen den Webservice-basierten RSAT und Ihrer älteren Infrastruktur vermittelt.

In den vielfältigen Lösungen zu Active Directory, die Sie in diesem Kapitel finden, werden primär die Active Directory-Cmdlets von Microsoft eingesetzt. Sie finden beinahe immer aber auch Vergleichscode, der zeigt, wie dieselbe Aufgabe mit nativem .NET-Code umgesetzt wird.

Spätestens wenn Sie sich für die Verwaltung lokaler Konten interessieren, wird dieser .NET-Ansatz in den Vordergrund gerückt, weil es zur Verwaltung lokaler Konten keine Cmdlets gibt.

TIPP

Auch von Drittanbietern existieren PowerShell-Erweiterungen für Active Directory. Die Firma Quest stellt beispielsweise bereits seit Jahren kostenfrei eine solche Erweiterung zur Verfügung (www.quest.com/powershell/), die den Vorzug hat, dass Sie weder Windows Server 2008 R2 noch Gatewaydienste benötigt und deshalb in den meisten Active Directory-Infrastrukturen sofort einsetzbar ist.

Active Directory-Cmdlets verwenden

Sie möchten die Microsoft-Cmdlets für Active Directory verwenden.

Lösung

Windows Server 2008 R2 enthält bereits die RSAT-Tools, die die Active Directory-Cmdlets enthalten. Auf anderen Betriebssystemen muss RSAT zuerst von Microsoft heruntergeladen und installiert werden. Suchen Sie dazu bei einer Internetsuchmaschine nach »RSAT download«.

Damit die Active Directory-Cmdlets tatsächlich auch verwendet werden können, muss das entsprechende Feature aktiviert werden. Unter Windows Server 2008 R2 stellen Sie dies mit folgendem Code sicher:

```
PS > Import-Module ServerManager
PS > Add-WindowsFeature RSAT-AD-PowerShell -IncludeAllSubFeature
Success Restart Needed Exit Code Feature Result
-----
True      No                Success {Active Directory-Modul...
```

Auf anderen Betriebssystemen muss das Feature manuell in der Systemsteuerung aktiviert werden. Sie finden es im Dialogfeld *Windows-Funktionen aktivieren oder deaktivieren* im Knoten *Remoteserver-Verwaltungstools/Rollenverwaltungstools/AD DS- und AD LDS-Tools/Active Directory-Modul für PowerShell*. Fehlt dieser Eintrag, wurden die RSAT-Tools noch nicht installiert.

Das Modul kann anschließend bei Bedarf in jeder PowerShell-Sitzung mit folgendem Befehl geladen werden:

```
PS > Import-Module ActiveDirectory
```

Die darin enthaltenen neuen Cmdlets zur Verwaltung von Active Directory listen Sie mit *Get-Command* auf:

```
PS > Get-Command -Module ActiveDirectory
```

CommandType	Name	Definition
Cmdlet	Add-ADComputerServiceAccount	Add-ADComputerService...
Cmdlet	Add-ADDomainControllerPasswo...	Add-ADDomainControlle...
Cmdlet	Add-ADFineGrainedPasswordPol...	Add-ADFineGrainedPass...
Cmdlet	Add-ADGroupMember	Add-ADGroupMember [-I...

Cmdlet	Add-ADPrincipalGroupMembership	Add-ADPrincipalGroupM...
Cmdlet	Clear-ADAccountExpiration	Clear-ADAccountExpira...
Cmdlet	Disable-ADAccount	Disable-ADAccount [-I...
Cmdlet	Disable-ADOptionalFeature	Disable-ADOptionalFea...
Cmdlet	Enable-ADAccount	Enable-ADAccount [-Id...
Cmdlet	Enable-ADOptionalFeature	Enable-ADOptionalFeat...
Cmdlet	Get-ADAccountAuthorizationGroup	Get-ADAccountAuthoriz...
Cmdlet	Get-ADAccountResultantPasswo...	Get-ADAccountResultan...
Cmdlet	Get-ADComputer	Get-ADComputer -Filde...
(...)		

Hintergrund

Das Modul *ActiveDirectory*, das Teil der RSAT-Tools ist, liefert alle für die Verwaltung von Benutzern im Active Directory erforderlichen Cmdlets. Um die Cmdlets nutzen zu können, muss das Modul zuerst mit *Import-Module* importiert werden. Die Cmdlets stehen danach in der aktuellen PowerShell-Sitzung zur Verfügung. Möchten Sie das Modul nicht jedes Mal manuell importieren, fügen Sie den Importbefehl in Ihr Profilskript ein.

Seit der Einführung von Windows Server 2008 R2 kommunizieren die Active Directory-Verwaltungstools nicht mehr über das DCOM-Verfahren, sondern verwenden Webservices. Sind Sie nicht an einer Domäne angemeldet oder verwendet Ihre Domäne weder Windows Server 2008 R2 als Domänencontroller noch ist ersatzweise ein Gatewaydienst installiert, erhalten Sie nach dem Import des *ActiveDirectory*-Moduls diese Meldung:

WARNUNG: Fehler beim Initialisieren des Standardlaufwerks: "Es wurde kein Standardserver gefunden, auf dem die Active Directory-Webdienste ausgeführt werden."

Setzen Sie keinen Rechner mit Windows Server 2008 R2 als Domänencontroller ein, muss in Ihrer Domäne zuerst ein Gatewaydienst installiert werden. Er »übersetzt« die via Webservice übermittelten Anfragen und leitet sie an ältere Domänencontroller weiter. Der Gatewaydienst kann kostenfrei hier heruntergeladen werden:

<http://www.microsoft.com/downloads/en/details.aspx?displaylang=en&FamilyID=008940c6-0296-4597-be3e-1d24c1cf0dda>

Ist Ihr Computer nicht an einer Domäne angemeldet oder kein Domänenmitglied, melden Sie sich über folgenden Umweg bei einem Domänencontroller an:

```
PS > New-PSDrive -PSProvider ActiveDirectory -Server 192.168.2.234 -Credential →
Tobias -Root "" -Name AD1 -formatType Canonical
```

Diese Zeile legt ein neues Laufwerk namens *AD1*: an und meldet Sie dazu im Beispiel an einem Domänencontroller mit der Adresse *192.168.2.234* an. Gelingt der Anmeldevorgang, brauchen Sie nur noch zu diesem Laufwerk zu wechseln und sind mit der angegebenen Domäne verbunden:

```
PS > cd AD1:
```

```
PS > get-aduser -filter 'name -like "A*"'
```

```
DistinguishedName : CN=Administrator,CN=Users,DC=powershell,DC=local
Enabled           : True
GivenName        :
Name             : Administrator
ObjectClass      : user
ObjectGUID       : 8d6ec7b1-8bbb-40f7-8d87-d754918ccba6
SamAccountName   : Administrator
SID              : S-1-5-21-4077880392-891536175-4210444282-500
Surname          :
UserPrincipalName :
```

Stehen Ihnen die RSAT-Tools und damit das *ActiveDirectory*-Modul erst gar nicht zur Verfügung, können Sie dennoch damit arbeiten. Voraussetzung ist, dass Sie über PowerShell Remoting auf einen Computer innerhalb der Domäne zugreifen können, auf dem die RSAT-Tools zur Verfügung stehen. Dies muss kein Server sein. Es genügt ein einfacher Client, solange dieser Mitglied der Domäne ist und über das *ActiveDirectory*-Modul verfügt.

In diesem Fall greifen Sie entweder interaktiv mit *Enter-PSSession* auf den Domänencomputer zu oder Sie blenden die Cmdlets des *ActiveDirectory*-Moduls über implizites Remoting in Ihre eigene Session ein.

Die folgenden Zeilen melden sich zuerst via PowerShell Remoting bei einem System an, das Mitglied in Active Directory ist und über die RSAT-Tools verfügt. Danach wird das Modul *ActiveDirectory* in die Remotesitzung geladen und anschließend die Cmdlets dieser Session in die eigene lokale Session importiert:

```
PS > $session = New-PSSession -ComputerName 192.168.2.234 -Credential Tobias
PS > Invoke-Command { Import-Module ActiveDirectory } -session $session
PS > Import-PSSession -Session $session -Module ActiveDirectory
```

ModuleType	Name	ExportedCommands
Script	tmp_1c6da115-119a-4426...	{Set-ADOrganizationalUnit, Get-A...

Anschließend können die Active Directory-Cmdlets genauso eingesetzt werden, als wären Sie auf dem eigenen Computer vorhanden. Tatsächlich werden die Cmdlets jedoch über implizites Remoting auf dem Remotesystem ausgeführt:

```
PS > Get-ADUser -filter *
```

```
RunspaceId       : e332954e-95bd-4abe-8762-4df7dd393f6d
DistinguishedName : CN=Administrator,CN=Users,DC=powershell,DC=local
Enabled          : True
GivenName        :
Name             : Administrator
ObjectClass      : user
ObjectGUID       : 8d6ec7b1-8bbb-40f7-8d87-d754918ccba6
SamAccountName   : Administrator
```

```
SID : S-1-5-21-4077880392-891536175-4210444282-500
Surname :
UserPrincipalName :
(...)
```

Diese Zugriffstechniken werden in Kapitel 18 (PowerShell Remoting) genauer beschrieben.

ACHTUNG Weil beim impliziten Remoting jedes Cmdlet einzeln auf dem Remotesystem ausgeführt wird, kann es zu Problemen bei zusammengesetzten Pipelines kommen. Die folgende Befehlsfolge würde beispielsweise ein neues Benutzerkonto anlegen und es dann aktivieren:

```
New-ADUser | Enable-ADUser
```

Greifen Sie über implizites Remoting auf diese Cmdlets zu, werden die Befehle einzeln nacheinander ausgeführt und die Ergebnisse jeweils zuerst zurück zu Ihrem Computer geleitet. Bei diesem Vorgang verändern sich die Objekte durch einen Prozess, der Serialisierung genannt wird. Die folgenden Befehle der Pipeline erkennen die Objekttypen also nicht mehr und können die Informationen nicht nahtlos weiterverarbeiten.

Mit Active Directory verbinden

Sie wollen eine Verbindung zu Active Directory herstellen, um anschließend auf Informationen von diesem Active Directory zuzugreifen oder beispielsweise dort neue Benutzer anzulegen.

Lösung

Wenn Sie das Modul *ActiveDirectory* importiert haben, meldet es sich automatisch an Ihrer aktuellen Domäne an. Sie müssen sich nur dann erneut anmelden, wenn Sie mit mehreren Domänen oder einer anderen Domäne als Ihrer augenblicklichen arbeiten wollen. Eine Anmeldung ist auch dann nötig, wenn Sie mit Ihrem Computer noch an gar keiner Domäne angemeldet sind.

Sie können sich dazu entweder mit den Parametern *-Server* und *-Credential* für jeden Cmdlet-Aufruf einzeln anmelden. Hierbei werden Ihre Anmeldeinformationen in der Pipeline an eventuell folgende Cmdlets weitergereicht, doch muss die Anmeldung für jeden neuen Pipelineaufruf erneut erfolgen:

```
PS > $cred = Get-Credential powershell\Tobias
PS > Get-ADUser -Server 192.168.2.234 -Credential $cred -filter *
```

Oder Sie legen ein oder mehrere neue virtuelle Laufwerke an und verbinden diese jeweils mit Active Directory. In diesem Fall verwenden alle Cmdlets die Anmeldedaten des gerade gewählten Laufwerks. Die folgenden beiden Zeilen legen zwei neue virtuelle Laufwerke namens »AD1« und »AD2« an und melden Sie jeweils unter anderer Identität an zwei verschiedenen Domänen an:


```
PS > New-PSDrive -PSProvider ActiveDirectory -Server 192.168.2.234 -Credential →  
powershell\Tobias -Root "" -Name AD1 -formatType Canonical  
PS > New-PSDrive -PSProvider ActiveDirectory -Server 192.168.4.113 -Credential →  
contoso\Frank -Root "" -Name AD2 -formatType Canonical
```

Wechseln Sie anschließend jeweils in das Laufwerk der Domäne, mit der Sie arbeiten wollen. Alle Active Directory-Cmdlets verwenden nun die Anmeldeinformationen des gewählten Laufwerks:

```
PS > Cd AD1:  
PS > Get-ADUser -filter *
```

Hintergrund

Die explizite Anmeldung an einer Domäne kommt häufig in drei Szenarien vor:

- **Externer Mitarbeiter** Sie sind als Berater in einem Unternehmen tätig und nicht an der Unternehmensdomäne angemeldet. Trotzdem möchten Sie auf die Domäne zugreifen und verfügen darin auch über ein gültiges Benutzerkonto, mit dem Sie sich anmelden können.
- **Forests und große Organisationen** Ihr Unternehmen verwaltet mehrere Domänen und Sie müssen sich von Fall zu Fall mit einer bestimmten Domäne verbinden
- **Replikation** Sie möchten selbst bestimmen, mit welchem Domänencontroller Sie verbunden werden, weil Sie Änderungen an Active Directory gezielt auf einem bestimmten Server starten möchten, um so die Replikationsgeschwindigkeit zu beeinflussen

Zwar können Sie sich bei jedem Aufruf eines Cmdlets neu anmelden. Ein sehr viel einfacherer und effizienterer Weg ist aber, für jede Domäne, mit der Sie arbeiten möchten, ein eigenes virtuelles Laufwerk anzulegen. Jetzt brauchen Sie nur noch von Fall zu Fall mit *cd* in das passende Laufwerk zu wechseln.

Möchten Sie Active Directory ohne Cmdlets direkt mit .NET-Methoden verwalten, erfolgt die Anmeldung über .NET. Die folgende Zeile meldet Sie an der aktuellen Domäne im Rootverzeichnis an und setzt voraus, dass Ihr Benutzerkonto bereits an einer Domäne angemeldet ist.

```
PS > $domain = [ADSI]''  
PS > $domain  
  
distinguishedName  
-----  
{DC=contoso,DC=local}
```

Um sich an einer beliebigen Domäne explizit anzumelden, beschaffen Sie sich mit *New-Object* ein Objekt des Typs *DirectoryServices.DirectoryEntry*. Anstelle des Rootordners können Sie sich auch mit einem beliebigen Container der Domäne verbinden. Die nächsten Zeilen demonstrieren, wie Sie sich an einem bestimmten Domänencontroller (hier IP-Adresse 192.168.2.234) mit der Organisationseinheit »Vertrieb« anmelden:

```
PS > $cred = Get-Credential powershell\Tobias
PS > $domain = New-Object DirectoryServices.DirectoryEntry →
('LDAP://192.168.2.234/OU=Vertrieb,DC=powershell,DC=local' →
,$cred.UserName, $cred.GetNetworkCredential().Password)
```

Sie können die Anmeldedaten auch vorgeben:

```
PS > $domain = new-object DirectoryServices.DirectoryEntry →
("LDAP://192.168.2.234","powershell\Tobias", "P@ssw0rd")
```

Möchten Sie sich an einer anderen Domäne anmelden und benötigen dort keine besondere Authentifizierung, weil Ihr aktuelles Benutzerkonto bei dieser Domäne Administratorrechte besitzt, verwenden Sie diese Zeile:

```
PS > $domain = [ADSI]'LDAP://testdomain'
```

ACHTUNG Die Angaben, die Sie für die Verbindung zur Domäne verwenden, müssen exakt so wie angegeben verwendet werden. Der Provider-Name *LDAP:* unterscheidet zwischen Groß- und Kleinschreibung. Es müssen normale Schrägstriche verwendet werden. Umgekehrte Schrägstriche sind nicht erlaubt. Die folgenden beiden Anweisungen schlagen deshalb fehl:

```
$domain = [ADSI]"ldap://10.10.10.1"           # FALSCH!
$domain = [ADSI]"LDAP:\\10.10.10.1"          # FALSCH!
```

Neues Benutzerkonto anlegen

Sie möchten ein neues Benutzerkonto in Active Directory anlegen.

Lösung

Verwenden Sie das Cmdlet *New-ADUser* und legen Sie fest, welche Eigenschaften im neuen Benutzerkonto hinterlegt werden sollen und wo das neue Benutzerkonto in Active Directory angelegt wird. Mindestens der Parameter *-Name* muss angegeben werden, aber außerdem sollte stets auch der Parameter *-SamAccountName* angegeben werden, weil dieser dem NetBIOS-Anmeldenamen des Kontos entspricht und andernfalls auf einen zufälligen Wert festgelegt würde. Geben Sie nur diese Minimalangaben an, wird das Konto im Container *Users* angelegt.

```
PS > New-ADUser -name Tweltner -SamAccountName Tweltner
```

Sie können den Erfolg sofort überprüfen, indem Sie mit dem virtuellen PowerShell-Laufwerk in diesen Container wechseln und seinen Inhalt anzeigen lassen:

```
PS > cd ad1:\powershell.local/users
PS > dir t*
```

Name	ObjectClass	DistinguishedName	CanonicalName
Tobias	user	CN=Tobias,CN=U...	powershell.local...
Tweltner	user	CN=Tweltner,CN...	powershell.local...

Möchten Sie einen Benutzer an einem anderen Ort anlegen, geben Sie den Parameter *–Path* an:

```
PS > New-ADUser -name 'Frank Mohrbach' -SamAccountName FMohrbach -Path →
      'OU=Vertrieb,DC=powershell,DC=local'
```

Name	ObjectClass	DistinguishedName	CanonicalName
Frank Mohrbach	user	CN=Frank Mohrba...	powershell.loca...
FrankW	user	CN=FrankW,OU=Ve...	powershell.loca...
Vertriebsmita...	organiz...	OU=Vertriebsmit...	powershell.loca...

Hintergrund

Werden neue Benutzer angelegt, sind diese zunächst deaktiviert. Erst wenn dem Konto ein Kennwort zugewiesen wurde, das den Komplexitätskriterien und sonstigen Voraussetzungen der Domäne genügt, oder wenn das Konto so konfiguriert wurde, dass sich der Inhaber bei der nächsten Anmeldung ein neues Kennwort zuweisen muss, kann es mit *Enable-ADAccount* aktiviert werden.

Die folgende Zeile legt ein neues Benutzerkonto mit Kennwort an und aktiviert das Konto anschließend sofort:

```
PS > New-ADUser -name 'Frank Mohrbach' -SamAccountName FMohrbach -Path →
      'OU=Vertrieb,DC=powershell,DC=local' -accountpassword ('P@ssw0rd123' | →
      ConvertTo-SecureString -asPlainText -Force) -passThru | Enable-ADAccount
```

Wichtig ist hierbei der Parameter *–passThru*, der dafür sorgt, dass das neu generierte Benutzerkonto von *New-ADUser* zurückgeliefert und so an das nächste Cmdlet in der Pipeline weitergereicht werden kann.

Möchten Sie, dass der Inhaber des neuen Kontos sich bei der nächsten Anmeldung selbst ein Kennwort zuweisen soll, hinterlegen Sie dennoch ein geeignetes Kennwort, weil das Konto sonst nicht aktiviert werden kann, und geben zusätzlich den Parameter *–ChangePasswordAtLogon* an:

```
PS > New-ADUser -name 'Frank Mohrbach' -SamAccountName FMohrbach -Path →
      'OU=Vertrieb,DC=powershell,DC=local' -accountpassword ('P@ssw0rd123' | →
      ConvertTo-SecureString -asPlainText -Force) -ChangePasswordAtLogon $true →
      -passThru | Enable-ADAccount
```

Möchten Sie ein neues Benutzerkonto mit reinen .NET-Methoden anlegen, gehen Sie so vor:

```
PS > $username = 'FrankW'
PS > $password = 'P@ssw0rd99'
PS > $description = 'mit PowerShell angelegter Benutzer'
PS > $container = [ADSI]'OU=Vertrieb,DC=powershell,DC=local'
PS >
PS > $user = $container.Create('user', "CN=$username")
PS > $user.SetInfo()
PS > $user.Description = $description
PS > $user.SetInfo()
PS > $user.SetPassword($password)
PS > $user.psbase.InvokeSet('AccountDisabled', $false)
PS > $user.SetInfo()
```

Viele neue Benutzerkonten anlegen

Sie möchten eine Liste mit Benutzerkonten in Active Directory anlegen.

Lösung

Speichern Sie die Liste als kommaseparierte Datei und weisen Sie den Spalten die Namen der Parameter zu, an die die Daten übergeben werden sollen. Anschließend kann die Liste mit *Import-CSV* eingelesen und sofort an *New-ADUser* weitergeleitet werden.

Legen Sie zum Beispiel diese Liste als Textdatei namens *users.csv* an oder exportieren Sie eine Liste aus Excel:

```
Name, SAMAccountName, Path, Description
TobiasWeltner, TWeltner, "OU=Vertrieb,DC=powershell,DC=local", Vertriebsleiter
FrankMueller, FMueller, "OU=Vertrieb,DC=powershell,DC=local", Mitarbeiter
Testkontol, test1, "CN=Users,DC=powershell,DC=local", Konto für Testzwecke
```

Lesen Sie die Daten anschließend ein und leiten Sie sie weiter an *New-ADUser*:

```
PS > Import-CSV c:\kurs1\users.csv | New-ADUser -accountpassword →
('P@ssw0rd123' | ConvertTo-SecureString -asPlainText -Force) →
-ChangePasswordAtLogon $true -passThru | Enable-ADAccount
```

TIPP

Falls Sie eine Benutzerliste mit Excel erstellt und als CSV-Datei exportiert haben, wird darin auf deutschen Systemen als Trennzeichen das Semikolon verwendet. Damit *Import-CSV* die Datei dennoch korrekt einliest, geben Sie den Parameter *-useCulture* an:

```
PS > Import-CSV c:\kurs1\users.csv -useCulture | New-ADUser -accountpassword →
('P@ssw0rd123' | ConvertTo-SecureString -asPlainText -Force) →
-ChangePasswordAtLogon $true
```

Speichern Sie die Liste im Dialogfeld *Speichern unter* mit dem Encoding *UTF8*, wenn deutsche Umlaute erhalten bleiben sollen.

Die Konten werden angelegt und jedem Konto ein Standardkennwort zugewiesen. Außerdem wird jedes Konto so konfiguriert, dass der Inhaber bei der ersten Anmeldung ein neues Kennwort auswählen muss.

Den Erfolg können Sie mit dieser Zeile überprüfen:

```
PS > cd \
PS > Import-CSV c:\kurs1\users.csv | Select-Object -expand SAMAccountName | Get-ADUser
```

Hintergrund

Viele Parameter der Active Directory-Cmdlets können ihre Daten auch von einem vorhergehenden Befehl der Pipeline empfangen. *Import-CSV* importiert den Inhalt einer CSV-Datei und verwandelt die darin enthaltenen Daten in Objekte. Jede Spaltenüberschrift wird zu einer Objekteigenschaft:

```
PS > Import-CSV c:\kurs1\users.csv
```

Name	SAMAccountName	Path	Description
----	-----	----	-----
TobiasWeltner	TWeltner	OU=Vertrieb,DC=p...	Vertriebsleiter
FrankMueller	FMueller	OU=Vertrieb,DC=p...	Mitarbeiter
Testkonto1	test1	CN=Users,DC=powe...	Konto f?r Testzw...

Es kommt bei der Erstellung der CSV-Datei also darauf an, die richtigen Spaltennamen zu wählen. Sie müssen genauso lauten wie die Parameter, die sie später empfangen sollen. Damit deutsche Umlaute beim Import nicht verloren gehen, sollte bei der Speicherung der Liste als Encoding *UTF8* gewählt werden. Verwendet die Datei nicht das Komma als Trennzeichen, legen Sie das Trennzeichen entweder mit dem Parameter *-delimiter* selbst fest, oder Sie geben den Parameter *-useCulture* an, damit das regionale Trennzeichen verwendet wird.

Im Beispiel wurde jedem Konto ein einheitliches Standardkennwort zugewiesen und mit *-ChangePasswordAtLogin* festgelegt, dass der Inhaber sich bei der nächsten Anmeldung selbst ein Kennwort zuweisen soll. Sie können aber auch individuelle Kennwörter zuweisen. Dazu erweitern Sie die CSV-Datei um eine Spalte namens *AccountPassword*.

```
Name, SAMAccountName, Path, Description, AccountPassword
TobiasWeltner, TWeltner, "OU=Vertrieb,DC=powershell,DC=local", Vertriebsleiter, P@ssw0rd1
FrankMueller, FMueller, "OU=Vertrieb,DC=powershell,DC=local", Mitarbeiter, P@ssw0rd2
Testkonto1, test1, "CN=Users,DC=powershell,DC=local", Konto für Testzwecke, P@$$w0rd
```

Weil der Parameter *AccountPassword* allerdings keinen Text erwartet, sondern einen sogenannten »SecureString«, muss diese Eigenschaft nun zuerst in einen *SecureString* umgewandelt werden, bevor das Ergebnis an *New-ADUser* weitergeleitet werden kann. Dazu verwenden Sie *ForEach-Object* und bearbeiten die eingelesenen Objekteigenschaften zuerst:

```
PS > Import-CSV c:\kurs1\users.csv | ForEach-Object { $_.AccountPassword = →
    $_.AccountPassword | ConvertTo-SecureString -asPlainText -Force; $_ } | →
    New-ADUser -accountpassword ('P@ssw0rd123' | ConvertTo-SecureString →
    -asPlainText -Force) -ChangePasswordAtLogon $true -passThru | Enable-ADAccount
```

Um die Beispielkonten der CSV-Datei wieder zu entfernen, greifen Sie zu dieser Zeile:

```
PS > Import-CSV c:\kurs1\users.csv | Select-Object -expand SAMAccountName | →
    Remove-ADUser
```

Benutzerkonto löschen

Sie möchten ein Benutzerkonto aus Active Directory entfernen.

Lösung

Greifen Sie mit *Get-ADUser* auf das Benutzerkonto zu und leiten Sie es dann mit *-passThru* an *Remove-ADUser* weiter:

```
PS > Get-ADUser -filter 'samaccountname -eq "fmohrbach"' | Remove-ADUser
```

Nach einer Sicherheitsabfrage wird das Konto unwiderruflich gelöscht.

Hintergrund

Weil das Löschen von Konten (oder anderen Objekten in Active Directory) unwiderruflich ist, erfolgt dabei standardmäßig eine Sicherheitsabfrage. Wollen Sie die Aufgabe unbeaufsichtigt durchführen, geben Sie den Parameter *-confirm* an und weisen ihm *\$false* zu:

```
PS > Get-ADUser -filter 'samaccountname -eq "fmohrbach"' | Remove-ADUser →
    -Confirm:$false
```

ACHTUNG Haben sich Fehler in Ihr Skript eingeschlichen und schalten Sie die automatische Bestätigung ab, kann das katastrophale Folgen haben. Die Bestätigung darf also nur abgeschaltet werden, wenn der Skriptcode zuvor ausführlich getestet und für unbedenklich erklärt wurde.

Mit reinen .NET-Methoden lassen sich beliebige Objekte in Active Directory löschen. Die folgenden Zeilen löschen das Benutzerkonto *Testuser* aus dem Container *Users* der aktuellen Domäne:

```
PS > $username = "Testuser"
PS > $ADsPath = "LDAP://CN=Users," + ([ADSI]"").distinguishedName
PS > $container = [ADSI]$ADsPath
PS > $container.Delete("User", "CN=$username")
```

Kennwort eines Benutzerkontos ändern

Sie möchten einem Benutzerkonto ein neues Kennwort zuweisen, zum Beispiel, weil der Inhaber seines vergessen hat oder es nicht mehr geheim ist. Oder Sie möchten das Kennwort eines eigenen Kontos ändern.

Lösung

Verwenden Sie *Set-ADAccountPassword* und geben Sie altes und neues Kennwort an. Die Kennwörter müssen dazu als *SecureString* vorliegen. Entweder erfragen Sie die Kennwörter direkt als *SecureString*:

```
PS > $old = Read-Host 'altes Kennwort' -asSecureString
PS > $new = Read-Host 'altes Kennwort' -asSecureString
PS > Set-ADAccountPassword TWeltner -old $old -new $new
```

Oder Sie konvertieren Klartextkennwörter in *SecureStrings* (nicht empfehlenswert, aber manchmal unvermeidbar):

```
PS > $old = 'P@ssw0rd1' | ConvertTo-SecureString -AsPlainText -Force
PS > $new = 'P@$$w0rd10' | ConvertTo-SecureString -AsPlainText -Force
PS > Set-ADAccountPassword TWeltner -old $old -new $new
```

Soll das Kennwort mit administrativer Gewalt neu gesetzt werden, geben Sie nur das neue Kennwort und dazu den Parameter *-Reset* an:

```
$new = 'P@$$w0rd10' | ConvertTo-SecureString -AsPlainText -Force
Set-ADAccountPassword TWeltner -new $new -Reset
```

Hintergrund

Auch von .NET aus lassen sich Kennwörter verwalten. Setzen Sie *ChangePassword()* ein, wenn das alte Kennwort bekannt ist, oder weisen Sie mit *SetPassword()* ein neues Kennwort zu. Hierbei sind Administratorrechte erforderlich.

Sprechen Sie das Konto an und verwenden Sie die Methode *ChangePassword()*, wenn Sie das alte Kennwort noch wissen, oder *SetPassword()*, um mit Administratorgewalt ein neues Kennwort zu setzen:

```
PS > $user = [ADSI]"LDAP://CN=Testuser1,CN=Users,DC=powershell,DC=local"
PS > $user.ChangePassword('altes Kennwort', 'neues Kennwort')
PS > $user.SetPassword('Neues Kennwort')
```

ACHTUNG Handelt es sich nicht um ein auf Active Directory basierendes Benutzerkonto, kann *SetPassword()* dazu führen, dass der Anwender den Zugriff auf seine persönlichen Kennwörter und Zertifikate verliert. Das kann im Extremfall dazu führen, dass der Anwender verschlüsselte E-Mails und verschlüsselte Dateien nicht mehr lesen kann.

SetPassword() und *ChangePassword()* sind nur erfolgreich, wenn das neue Kennwort den Komplexitätskriterien entspricht.

Benutzerkonto suchen

Sie möchten ein Benutzerkonto ansprechen, wissen aber nicht genau, wo es sich in Active Directory befindet.

Lösung

Verwenden Sie *Get-ADUser* und geben Sie entweder die exakte Identität an (*SamAccountName*, *SID*, *DN* oder *GUID*):

```
PS > Get-ADUser TWeltner
PS > Get-ADUser S-1-5-21-4077880392-891536175-4210444282-1126
PS > Get-ADUser 'CN=TobiasWeltner,OU=Vertrieb,DC=powershell,DC=local'
PS > Get-ADUser 765245ce-8366-4dbd-aa21-e8a0f1f015f1
```

Sie erhalten dann entweder genau das gesuchte Konto oder keines.

Alternativ verwenden Sie den Parameter *-filter*, um nach anderen Kriterien oder mit Platzhalterzeichen zu suchen:

```
PS > # alle Konten, deren SAMAccount-Name mit "T" beginnt:
PS > Get-ADUser -filter 'samaccountname -like "t*"'
PS >
PS > # alle Konten, deren E-Mail-Adresse auf "email.de" endet:
PS > Get-ADUser -filter 'emailaddress -like "*email.de"'
PS >
PS > # wie vor, aber nur Konten, in deren Name "mann" vorkommt:
PS > Get-ADUser -filter '(emailaddress -like "*email.de") -and →
    (name -like "*mann*")'
PS >
PS > # alle Konten mit E-Mail-Adresse:
PS > Get-ADUser -filter 'emailaddress -like "*"'
PS >
PS > # alle Konten ohne E-Mail-Adresse:
PS > Get-ADUser -filter 'emailaddress -notlike "*"'
PS >
PS > # alle Konten, die ihr Kennwort neu setzen müssen:
PS > Get-ADUser -Filter 'pwdLastSet -eq 0'
PS >
PS > # alle Konten ohne Beschreibung:
PS > Get-ADUser -Filter 'description -notlike "**'
```


Jetzt erhalten Sie je nach Auswahlkriterien beliebig viele Resultate. Active Directory liefert allerdings als Vorgabe nur maximal 1.000 Ergebnisse. Wollen Sie mehr, setzen Sie *–ResultSetSize \$null*.

Hintergrund

Benutzerkonten werden nur über vier Kriterien eindeutig identifiziert, nämlich den *SAM-Account*-Namen, die Sicherheitskennung (SID), den DN (Distinguished Name) und die GUID (Globally Unique Identifier). Als Vorgabe findet *Get-ADUser* Benutzerkonten nur, wenn Sie eine eindeutige Identity angeben.

TIPP

Die GUID eines Benutzerkontos ist die zuverlässigste Identity. Während die anderen Kriterien zwar eindeutig sind, aber geändert werden können, bleibt die GUID für den gesamten Lebenszyklus eines Benutzerkontos gleich, ändert sich also beispielsweise nicht, wenn der Mitarbeiter heiratet und einen neuen Namen erhält oder in eine andere Abteilung umzieht.

Get-ADUser startet seine Suche entweder im Wurzelverzeichnis von Active Directory oder im Wurzelverzeichnis des virtuellen PowerShell-Laufwerks, wenn Sie dorthin gewechselt haben. Möchten Sie den Start der Suche eindeutig festlegen, verwenden Sie den Parameter *–SearchBase*. Legen Sie *–SearchBase* auf einen leeren Text fest, wird im globalen Katalog gesucht, sofern vorhanden. Mit *–SearchScope* legen Sie fest, ob rekursiv gesucht werden soll (Vorgabe) oder nicht.

Auch mit .NET-Code können Benutzerkonten gesucht werden. Die Funktion *Get-User* sucht den Benutzernamen beispielsweise in der *SamAccountName*-Eigenschaft:

```
Function Get-User($UserName)
{
    $searcher = New-Object DirectoryServices.DirectorySearcher([ADSI] "")
    $searcher.filter = "(&(objectClass=user)(sAMAccountName= $UserName))"
    $searcher.findall() | Foreach-Object { $_.GetDirectoryEntry() }
}
```

PS > **Get-User T***

Auch hier können weitere Parameter konfiguriert werden, mit denen Sie bestimmen, ab wo und wie gesucht werden soll und wie viele Ergebnisse maximal zurückgeliefert werden:

```
Function Get-User ($UserName)
{
    $searcher = New-Object DirectoryServices.DirectorySearcher([ADSI] "")
    $searcher.filter = "(&(objectClass=user)(sAMAccountName= $UserName))"
    $Searcher.CacheResults = $true
    $Searcher.SearchScope = "Subtree"
    $Searcher.PageSize = 1000
    $searcher.findall() | Foreach-Object { $_.GetDirectoryEntry() } | →
        Select-Object *
}
```

TIPP

Setzen Sie *PageSize* auf 1.000 oder weniger, erhalten Sie mehr als 1.000 Ergebnisse. Das Limit von 1.000 Ergebnissen wird damit aufgehoben, weil nun die Ergebnisse in Paketen von 1.000 oder weniger Einträgen geliefert werden.

Möchten Sie von .NET aus ein Active Directory-Konto über dessen GUID ansprechen, geschieht dies so:

```
PS > $object = [ADSI]"LDAP://<GUID=7243bdf88f3c2348aacb7e379f33ccc9>"
```

Mit externer Authentifizierung gehen Sie so vor:

```
PS > $ldap = 'LDAP://192.168.2.106/<GUID=7243bdf88f3c2348aacb7e379f33ccc9>'
PS > $user = 'contoso\Tobias'
PS > $password = 'P@ssw0rd'
PS > $objekt = New-Object DirectoryServices.DirectoryEntry($ldap, $user, →
    $password)
```

Wollen Sie die GUID eines Kontos über .NET auslesen, greifen Sie auf dessen Eigenschaft *GUID* zu:

```
PS > $user = [ADSI]"LDAP://CN=Administrator,CN=Users,DC=powershell,DC=local"
PS > $user.GUID
7243bdf88f3c2348aacb7e379f33ccc9
```

Mit LDAP-Filtern suchen

Sie möchten alle Benutzerkonten finden, die nach einem bestimmten Stichtag angelegt wurden, oder alle Computerkonten, die keine Domänencontroller sind, oder Sie wollen sonstige spezialisierte Abfragen ausführen.

Lösung

Formulieren Sie die Abfrage in der LDAP-Abfragesprache. Bei dieser Abfragesprache müssen die Eigenschaften, in denen Sie suchen wollen, mit den nativen Attributnamen von Active Directory übereinstimmen. Sie dürfen also die E-Mail-Adresse nicht in der Eigenschaft *EmailAddress* suchen, sondern in *mail*.

Um alle Benutzerkonten zu finden, bei denen die Eigenschaft *mail* nicht ausgefüllt ist, formulieren Sie:

```
PS > Get-ADUser -LDAPFilter '(!mail=*)'
```

Um Benutzer mit einem bestimmten Vor- und Zunamen zu finden, kombinieren Sie mehrere Kriterien:

```
PS > Get-ADUser -LDAPFilter '(&(sn=Weltner)(givenName=Tobias))'
```

Es kommt also im Grunde nur darauf an, die passenden Objekteigenschaften zu kennen, die eine bestimmte Konfiguration beschreiben. Möchten Sie alle Konten sehen, die Einwählerlaubnis haben, ist beispielsweise die Eigenschaft *msNPAllowDialin* zuständig:

```
PS > Get-ADUser -ldapFilter '(msNPAllowDialin=TRUE)'
```

Die folgende Abfrage würde alle Konten finden, die bei der nächsten Anmeldung ihr Kennwort neu setzen müssen:

```
PS > Get-ADUser -LDAPFilter '(pwdLastSet=0)'
```

Möchten Sie prüfen, ob eine bestimmte Bedingung *nicht* zutrifft, setzen Sie ein Ausrufezeichen davor. So könnten Sie Konten finden, bei denen beispielsweise die *Description*-Eigenschaft leer ist:

```
PS > Get-ADUser -LDAPFilter '(!description=*)'
```

Auch numerische und Zeitvergleiche sind möglich. Der folgende Filter liefert alle Objekte, die nach dem 17.2.2011 angelegt worden sind:

```
PS > Get-ADUser -LDAPFilter '(whenCreated>20110217000000.0Z)'
```

In Active Directory steht der Code *1.2.840.113556.1.4.803* für eine binäre »Und«-Verknüpfung und *1.2.840.113556.1.4.804* für eine binäre »Oder«-Verknüpfung. Der folgende Filter liefert alle deaktivierten Konten:

```
PS > Get-ADUser -LDAPFilter '(userAccountControl:1.2.840.113556.1.4.803:=2)'
```

Wollen Sie stattdessen alle Konten sehen, die ein nicht ablaufendes Kennwort haben, prüfen Sie auf Bit 15 (65536):

```
PS > Get-ADUser -LDAPFilter '(userAccountControl:1.2.840.113556.1.4.803:=65536)'
```

Verwenden Sie den Wert 32, wenn Sie alle Konten auflisten wollen, die kein Kennwort haben müssen. Und denken Sie an das Ausrufezeichen, das das Ergebnis umdreht. Möchten Sie alle Konten sehen, die ein Kennwort besitzen müssen, schreiben Sie ein Ausrufezeichen vor die Bedingung:

```
PS > Get-ADUser -LDAPFilter '(!userAccountControl:1.2.840.113556.1.4.803:=32)'
```

ACHTUNG Ihr LDAP-Filter liefert nur solche Objekttypen zurück, die das zugrunde liegende Cmdlet auch liefern kann. Führen Sie einen LDAP-Filter mit *Get-ADUser* aus, erhalten Sie stets nur Benutzerkonten. Aber auch mit *Get-ADGroup* und *Get-ADComputer* lassen sich nützliche LDAP-Filter konzipieren.

So findet der folgende Filter alle Verteilergruppen, zumindest dann, wenn Sie wissen, dass *groupType* ein spezielles Bit besitzt, das bei Verteilergruppen nicht gesetzt ist:

```
PS > Get-ADGroup -LDAPFilter '(!groupType:1.2.840.113556.1.4.803:=2147483648)'
```

Analog finden Sie alle Computer, die keine Domänencontroller sind, weil diesen in der Eigenschaft *userAccountControl* ein entsprechendes Bit fehlt:

```
PS > Get-ADComputer -LDAPFilter '→  
'(!userAccountControl:1.2.840.113556.1.4.803:=8192)'
```

Neue Organisationseinheit anlegen

Sie möchten eine neue Organisationseinheit in Active Directory anlegen.

Lösung

Verwenden Sie *New-ADOrganizationalUnit* und geben Sie einen Namen an. Ohne weitere Parameter wird die Organisationseinheit auf Stammebene der Domäne angelegt:

```
PS > New-ADOrganizationalUnit 'Vertrieb'
```

Möchten Sie die Organisationseinheit an einem anderen Ort anlegen, geben Sie mindestens den Parameter *-Path* an. Über weitere Parameter können auch die übrigen Informationen einer Organisationseinheit gesetzt werden:

```
PS > New-ADOrganizationalUnit 'Nord' -Path 'OU=Vertrieb,DC=powershell,DC=local' →  
-Description 'Nordmitarbeiter'
```

Hintergrund

Mit *Get-ADOrganizationalUnit* können Sie den Erfolg überprüfen. Die nächste Zeile liefert die Namen sämtlicher Organisationseinheiten:

```
PS > Get-ADOrganizationalUnit -filter * | Select-Object Name
```

Auch über .NET lassen sich Organisationseinheiten anlegen. Nehmen Sie wie im Abschnitt »Mit Active Directory verbinden« ab Seite 456 beschriebenen Kontakt zur Domäne auf und legen Sie darin ein neues Objekt vom Typ *OrganizationalUnit* an:

```
PS > $domain = [ADSI]''  
PS > $OU = $domain.Create("OrganizationalUnit", "OU=Vertrieb")  
PS > $OU.Put("Description", "Vertrieb-Stamm")  
PS > $OU.Put("wwwHomePage", "http://firma.de/vertrieb.htm")  
PS > $OU.SetInfo()
```

Mit den Active Directory-Cmdlets legen Sie eine Organisationseinheit so an:

```
PS > New-ADOrganizationalUnit -name Vertrieb -DisplayName Vertrieb →  
-Description 'Vertrieb-Stamm'
```

Die Organisationseinheit wird so im Stamm der Domäne angelegt. Möchten Sie eine Organisationseinheit an anderer Stelle anlegen, geben Sie den Ort an. Die nächste Zeile legt eine weitere Organisationseinheit in der soeben angelegten Organisationseinheit an:

```
PS > New-ADOrganizationalUnit -name Vertriebsmitarbeiter -DisplayName →  
Vertriebsmitarbeiter -Description 'Vertrieb-Mitarbeiter' -Path →  
'OU=Vertrieb,DC=powershell,DC=local'
```

Eigenschaften eines Benutzerkontos lesen

Sie möchten einzelne oder alle Eigenschaften (Attribute) eines Benutzerkontos lesen.

Lösung

Greifen Sie mit *Get-ADUser* auf das Benutzerkonto zu und legen Sie mit *-Properties* fest, welche Eigenschaften (Attribute) Sie sehen möchten. Geben Sie einen Stern an, werden sämtliche Attribute angezeigt:

```
PS > Get-ADUser TWeltner -Properties *
```

Wollen Sie bestimmte Eigenschaften sehen, müssen diese kommasepariert komplett angegeben werden.

```
PS > Get-ADUser TWeltner -Properties mail, description  
  
Description           : Mein Konto  
DistinguishedName     : CN=TobiasWeltner,OU=Vertrieb,DC=powershell,DC=local  
Enabled               : True  
GivenName             :  
mail                 : tobias.weltner@email.de  
Name                  : TobiasWeltner  
ObjectClass           : user  
ObjectGUID            : 765245ce-8366-4dbd-aa21-e8a0f1f015f1  
SamAccountName        : TWeltner  
SID                   : S-1-5-21-4077880392-891536175-4210444282-1126  
Surname               :  
UserPrincipalName     :
```

Weil *Get-ADUser* stets auch die Basiseigenschaften mit einschließt, kann man das Ergebnis anschließend an *Select-Object* weiterleiten, um tatsächlich nur die wirklich benötigten Informationen zu sehen:

```
PS > Get-ADUser Tweltner -Properties mail, description | Select-Object name, →
mail, description
```

name	mail	description
----	----	-----
TobiasWeltner	tobias.weltner@email.de	Mein Konto

Hintergrund

Auch von .NET aus kann man die Eigenschaften eines Kontos sichtbar machen, indem man es an *Select-Object ** weiterleitet:

```
PS > $user = [ADSI]'LDAP://CN=TobiasWeltner,OU=Vertrieb,DC=powershell,DC=local'
PS > $user | Select-Object *
```

Weil es sich dabei aber um Rohdaten handelt, sind ihre Inhalte häufig nicht mit vertretbarem Aufwand zu interpretieren:

```
PS > $user | Select-Object *log*
```

lastLogoff	lastLogon	logonCount
-----	-----	-----
{System.__ComObject}	{System.__ComObject}	{0}

Die Ergebnisse von *Get-ADUser* dagegen sind bereits in lesbare Formate konvertiert und außerdem vollständig:

```
PS > Get-ADUser Tobias -Properties * | Select-Object *log*
```

```
BadLogonCount      : 0
lastLogoff         : 0
lastLogon          : 129423284582084228
LastLogonDate      : 16.02.2011 12:14:18
lastLogonTimestamp : 129423284582084228
logonCount         : 2
LogonWorkstations  :
MNSLogonAccount    : False
SmartcardLogonRequired : False
```

Es kann allerdings vorkommen, dass Eigenschaften wie *LastLogonTimestamp* eine ungewöhnliche Maßeinheit besitzen, nämlich beispielsweise ein Datum in Form von Systemticks (Taktanzahl) angeben. Systemticks können über .NET in ein lesbares Datum umgewandelt werden:

```
PS > $ticks = Get-ADUser Tobias -Properties * | Select-Object -expand →
LastLogonTimestamp
PS > [DateTime]::FromFileTime($ticks)
```

```
Mittwoch, 16. Februar 2011 12:14:18
```

Wie sich zeigt, ist diese Umwandlung bereits von *Get-ADUser* geleistet worden, denn dieses Datum findet sich in der Eigenschaft *LastLogonDate*.

Eigenschaften eines Benutzerkontos ändern

Sie möchten nachträglich Änderungen an einem Benutzerkonto vornehmen. Zum Beispiel möchten Sie einem Benutzerkonto eine neue E-Mail-Adresse zuweisen.

Lösung

Greifen Sie mit *Get-ADUser* auf das Benutzerkonto zu und leiten Sie es an *Set-ADUser* weiter. Mit den Parametern von *Set-ADUser* setzen Sie die neuen Informationen:

```
PS > Get-ADUser Tweltner | Set-ADUser -EmailAddress tobias.weltner@email.de
```

Möchten Sie beispielsweise, dass ein Konto nach einem bestimmten Stichtag abläuft, verwenden Sie *-AccountExpirationDate*:

```
PS > Get-ADUser Tweltner | Set-ADUser -AccountExpirationDate (Get-Date 4.8.2011)
```

Wollen Sie Informationen löschen, verwenden Sie *-Clear*. Dazu müssen die Attributnamen der entsprechenden Active Directory-Attribute angegeben werden. Die folgende Zeile entfernt E-Mail-Adresse und Beschreibung:

```
PS > Get-ADUser Tweltner | Set-ADUser -Clear mail, Description
```

Möchten Sie Informationen ändern oder eintragen, für die es keinen Parameter gibt, verwenden Sie *-Add* oder *-Replace* und geben die zu ändernden Attribute als Schlüssel-Wert-Paare in Form eines Hashtables an. Die folgende Zeile fügt E-Mail-Adresse und Beschreibung ein:

```
PS > Get-ADUser Tweltner | Set-ADUser -Add @{mail='tobias.weltner@email.de'; →  
description='Mein Konto'}
```

Hintergrund

Für die gebräuchlichsten Eigenschaften eines Benutzerkontos liefert *Set-ADUser* bereits Parameter, denen lediglich der neue Wert zugewiesen zu werden braucht. Da Active Directory erweiterbar ist, können Parameter niemals alle denkbaren Einstellungen eines Benutzerkontos verwalten. Deshalb kann man diese über *-Add* und *-Replace* auch in Form eines Hashtables hinterlegen. Der Schlüssel entspricht dabei dem Attributnamen und der Wert seinem Inhalt.

Auch .NET kann auf die Attribute zugreifen und sie ändern. So ändern Sie zum Beispiel die Beschreibung des lokalen Administratorkontos:

```
PS > $computer = [ADSI]"WinNT://$env:computername,computer"  
PS > $user = $computer.psbases.children.Find("Administrator", "User")  
PS > $user.Description = 'Eine neue Beschreibung'  
PS > $user.SetInfo()
```

Und so hinterlegen Sie beim Active Directory-Benutzerkonto *Testuser* eine neue Telefonnummer in der Eigenschaft *telephoneNumber*:

```
PS > $user = [ADSI]"LDAP://CN=Testuser,CN=Users,DC=contoso,DC=local"  
PS > $user.telephoneNumber = '0511 1234567-89'  
PS > $user.SetInfo()
```

.NET kennt drei verschiedene Arten, Objekteigenschaften zu lesen und zu schreiben. Im Beispiel wurde die gebräuchlichste verwendet, nämlich die Punktnotation. Sie ist allerdings auch die fehlerträchtigste und darf nicht verwendet werden, wenn Sie neue Objekte anlegen.

Die zweite Variante sind die Methoden *Get()* und *Put()*. Sie könnten die Beispiele von oben entsprechend umformulieren:

```
PS > $user = [ADSI]"LDAP://CN=Testuser,CN=Users,DC=contoso,DC=local"  
PS > $user.Put('telephoneNumber', '0511 1234567-89')  
PS > $user.SetInfo()
```

Allerdings funktionieren auch *Put()* und *Get()* mit neu angelegten Objekten unzuverlässig. Der einzig sichere Weg, Eigenschaften beliebiger ADSI-Objekte zu lesen und zu ändern, verwendet die Methoden *InvokeGet()* und *InvokeSet()* des zugrunde liegenden *PSBase*-Objekts:

```
PS > $user = [ADSI]"LDAP://CN=Testuser,CN=Users,DC=contoso,DC=local"  
PS > $user.PSBase.InvokeSet('telephoneNumber', '0511 1234567-89')  
PS > $user.SetInfo()
```

Für besondere Fälle gibt es eine Zusatzmöglichkeit, Eigenschaften zu ändern: die Methode *PutEx()*. Sie kann zum Beispiel Eigenschaften löschen:

```
PS > $user.PutEx(1, "Description", 0)  
PS > $user.SetInfo()
```

Die Eigenschaft wird komplett entfernt und sollten Sie anschließend versuchen, sie zu lesen, erhalten Sie einen Fehler:

```
PS > $user.Get("Description")
```

Welche Aufgabe *PutEx()* genau ausführt, legen Sie mit dem ersten Argument als Zahl fest:

Zahl	Aufgabe
1	Eigenschaft löschen
2	Eigenschaft ersetzen
3	Eigenschaft erweitern
4	Eigenschaft teilweise löschen

Tabelle 16.1 Arbeitsweise von *PutEx()*

Mit *PutEx()* kann man zum Beispiel Eigenschaften handhaben, die mehrere Werte beinhalten, also Felder sind. Die folgenden Zeilen legen einige Telefonnummern für ein Benutzerkonto fest:

```
PS > $user.PutEx(2, "otherHomePhone", @("123", "456", "789"))
PS > $user.SetInfo()
```

PutEx() kann später weitere Telefonnummern zur vorhandenen Liste hinzufügen:

```
PS > $user.PutEx(3, "otherHomePhone", @("555"))
PS > $user.SetInfo()
```

Ebenso gut kann *PutEx()* ausgewählte Telefonnummern aus der Liste streichen:

```
PS > $user.PutEx(4, "otherHomePhone", @("456", "789"))
PS > $user.SetInfo()
```

Gruppenmitgliedschaft eines Benutzerkontos auflisten

Sie möchten wissen, in welchen Gruppen ein bestimmtes Benutzerkonto Mitglied ist.

Lösung

Greifen Sie auf das Benutzerkonto zu und rufen Sie dessen *memberOf*-Eigenschaft ab:

```
PS > Get-ADUser Tobias -Properties memberof | Select-Object -expand memberof
```

Hintergrund

Alle direkten Mitgliedschaften werden in der Eigenschaft *memberOf* vermerkt. Lediglich die Standardgruppe der Domäne (normalerweise »Domänen-Benutzer«) wird darin nicht aufgeführt.

Diese Information kann man auch direkt über .NET Framework abrufen. Für die aktuelle Domäne erfahren Sie die Gruppenmitgliedschaften des Administratorkontos so:

```
PS > $ADsPath = "LDAP://CN=Administrator,CN=Users," + ([ADSI] "").DistinguishedName
PS > $user = [ADSI]$ADsPath
PS > $user.memberOf
CN=Richtlinien-Ersteller-Besitzer,CN=Users,DC=contoso,DC=local
CN=Domänen-Admins,CN=Users,DC=contoso,DC=local
CN=Organisations-Admins,CN=Users,DC=contoso,DC=local
CN=Schema-Admins,CN=Users,DC=contoso,DC=local
CN=Administratoren,CN=Builtin,DC=contoso,DC=local
```

Gruppenmitglieder auslesen

Sie möchten wissen, wer Mitglied in einer bestimmten Gruppe ist.

Lösung

Verwenden Sie *Get-ADGroupMember* und geben Sie den Namen der gewünschten Gruppe an. Die folgende Zeile liefert alle Mitglieder der Gruppe »Domänen-Admins«:

```
PS > Get-ADGroupMember Domänen-Admins | Select-Object name, distinguishedname

name                distinguishedname
----                -
Administrator       CN=Administrator,CN=Users,DC=powersh...
(...)
```

Hintergrund

Falls Sie eine sonderbare »format default«-Fehlermeldung erhalten sollten, greifen Sie vermutlich gerade nicht auf Ihre Standarddomäne zu. Der Fehler entsteht, weil die von Active Directory gelieferten Ergebnisse auf Ihrem System formatiert werden sollen und dafür Berechtigungen fehlen. Sie umgehen das Problem, indem Sie die Daten an *Select-Object* weiterleiten.

Über .NET erfahren Sie Gruppenmitglieder so:

```
PS > $ADsPath = "LDAP://CN=Domänen-Admins,CN=Users," + →
([ADSI] "").DistinguishedName
PS > $group = [ADSI]$ADsPath
PS > $group.member
CN=Tobias Weltner,CN=Users,DC=contoso,DC=local
CN=Tobias Weltner,OU=Softgrid,DC=contoso,DC=local
CN=Administrator,CN=Users,DC=contoso,DC=local
```

Benutzerkonto zu einer Gruppe hinzufügen

Sie möchten ein Benutzerkonto als neues Mitglied in eine Gruppe aufnehmen.

Lösung

Fügen Sie den Benutzer mit *Add-ADGroupMember* der Gruppe hinzu. Die folgende Zeile fügt den Benutzer »TWeltner« in die Gruppe »Domänen-Admins« ein:

```
PS > Add-ADGroupMember 'Domänen-Admins' 'TWeltner'
```

Hintergrund

Auch über .NET ist die Mitgliedverwaltung von Gruppen möglich. Die folgenden Zeilen fügen den Benutzer *TWeltner* aus dem Container *Users* in die Gruppe der *Domänen-Admins* hinzu:

```
PS > $AdsPathUser = "LDAP://CN=TWeltner,CN=Users," + ([ADSI]"").DistinguishedName
PS > $AdsPathGroup = "LDAP://CN=Domänen-Admins,CN=Users," + ([ADSI]"").DistinguishedName
PS > $user = [ADSI]$AdsPathUser
PS > $group = [ADSI]$AdsPathGroup
PS > # Möglichkeit 1:
PS > $group.Add($user.psbase.Path)
PS > # Möglichkeit 2:
PS > $group.Add("LDAP://" + $user.distinguishedName)
```

Wie Sie sehen, müssen Sie bei diesem Ansatz die genauen *ADsPaths* von Benutzer und Gruppe kennen.

Jede Gruppe verfügt über eine *Add()*-Methode. Sie verlangt den *ADsPath*-Namen des Objekts, das neues Mitglied in der Gruppe werden soll. Die Änderung wird wie immer bei Methoden sofort wirksam und erfordert kein *SetInfo()*.

Alternativ können Sie die Gruppenmitgliedschaft auch über die *Member*-Eigenschaft ändern:

```
PS > $group.Member += $user.distinguishedName
PS > $group.SetInfo()
```

Wegen der schlecht implementierten Punkt-Notation in PowerShell ist davon aber abzuraten.

Benutzerkonto aus einer Gruppe entfernen

Sie möchten einen Benutzer aus einer Gruppe entfernen, um ihm die mit der Gruppenmitgliedschaft verbundenen Rechte und Pflichten zu entziehen.

Lösung

Entfernen Sie den Benutzer mit *Remove-ADGroupMember* aus einer Gruppe. Die folgende Zeile entfernt den Benutzer »TWeltner« aus der Gruppe der Domänen-Admins:

```
PS > Remove-ADGroupMember 'Domänen-Admins' 'TWeltner'
```

Hintergrund

Anders als beim Hinzufügen von Gruppenmitgliedern erfolgt beim Entfernen eine Sicherheitsabfrage. Ist diese unerwünscht, kann sie mit *-Confirm* abgeschaltet werden:

```
PS > Remove-ADGroupMember 'Domänen-Admins' 'TWeltner' -Confirm:$false
```

Von .NET aus entfernen Sie Gruppenmitglieder auf diese Weise:

```
PS > $AdsPathGroup = "LDAP://CN=Domänen-Admins,CN=Users," + →
([ADSI] "").DistinguishedName
PS > $AdsPathUser = "LDAP://CN=Testuser1,CN=Users," + ([ADSI] "").DistinguishedName
PS > $user = [ADSI]$AdsPathUser
PS > $group = [ADSI]$AdsPathGroup
PS > $group.Remove($user.psbases.path)
PS > # Alternative 1:
PS > $group.Remove("LDAP://" + $user.distinguishedName)
PS > # Alternative 2:
PS > $group.Remove("LDAP://CN=Testuser1,CN=Users,DC=contoso,DC=local")
```

Auf ein lokales Benutzerkonto zugreifen

Sie wollen auf ein vorhandenes lokales Benutzerkonto zugreifen, zum Beispiel, um dessen Eigenschaften zu ändern oder das Kennwort neu zu setzen.

Lösung

Da es kein Cmdlet zum Lesen von lokalen Benutzerkonten gibt, kann mithilfe von .NET Framework die Funktionalität nachgerüstet werden. Die folgende Funktion *Get-LocalUser* liefert als Standard das aktuelle Benutzerkonto, kann aber auch andere Benutzerkonten auf dem eigenen oder einem anderen Computer abrufen:

```
Function Get-LocalUser {
    Param(
        $username = $env:username,
        $computername = $env:userdomain
    )
    $computer = [ADSI]"WinNT://$computername,computer"
    $computer.psbases.children.Find($username, "User")
}
```

```
PS > Get-LocalUser | Select-Object *
```

```
UserFlags           : {66081}
MaxStorage           : {-1}
PasswordAge          : {34906954}
```


Mitglieder

Administrator

FHermann

Tobias

Training

Der Befehl wurde erfolgreich ausgeführt.

Auch mit der folgenden Funktion *Get-LocalGroup* greifen Sie auf eine lokale Gruppe zu und können beispielsweise deren *ADsPath* ermitteln oder Eigenschaften wie die Beschreibung lesen oder ändern:

```
function Get-LocalGroup {
    param(
        $groupname = 'Administratoren',
        $computername = $env:userdomain
    )
    $computer = [ADSI]"WinNT://$computername,computer"
    $computer.psbases.children.Find($groupname, "Group")
}
```

PS > **Get-LocalGroup Administratoren | Select-Object ***

```
groupType      : {4}
Name           : {Administratoren}
Description    : {Administratoren haben uneingeschränkten
                  Vollzugriff auf den Computer bzw. die Domäne.}
objectSid      : {1 2 0 0 0 0 0 5 32 0 0 0 32 2 0 0}
(...)
```

Um die Mitglieder der Gruppe aufzulisten, ermitteln Sie den *ADsPath* der gewünschten Gruppe und übergeben diesen dann an die folgende Funktion *Get-LocalGroupMembers*:

```
function Get-LocalGroupMembers($ADsPath= →
    "WinNT://$env:computername/Administratoren") {
    $group = [ADSI]$ADsPath
    $group.psbases.invoke("Members") |
        Foreach-Object { $_.GetType().InvokeMember('ADsPath', →
            'GetProperty', $null, $_, $null)}
}
```

PS > **\$ADsPath = Get-LocalGroup Administratoren | Select-Object -expand Path**

PS > **\$ADsPath**

WinNT://WORKGROUP/DEMO5/Administratoren

PS > **Get-LocalGroupMembers \$ADsPath**

WinNT://WORKGROUP/DEMO5/Administrator

WinNT://WORKGROUP/DEMO5/demo

WinNT://WORKGROUP/DEMO5/Tobias

WinNT://WORKGROUP/DEMO5/Training

Hintergrund

Get-LocalGroup greift per Default auf die lokale Gruppe der Administratoren zu und nutzt hierfür die Low-Level-Funktionen des ADSI (*Active Directory Service Interfaces*). Möchten Sie auf eine andere Gruppe auf dem eigenen oder einem anderen Computer zugreifen, verwenden Sie die Parameter *-groupname* und/oder *-computername*.

Get-LocalGroupMembers erwartet den *ADsPath* einer beliebigen Gruppe und listet dann die *ADsPaths* der darin enthaltenen Mitglieder auf. Mithilfe des *ADsPaths* werden Benutzer und Gruppen stets eindeutig beschrieben und über den *ADsPath* erhalten Sie von ADSI auch Zugriff auf das jeweilige Objekt:

```
PS > $objekt = [ADSI]$ADsPath
```

Möchten Sie lediglich die Mitglieder einer lokalen Gruppe ermitteln, kann das Befehlszeilentool *net.exe* eine einfache Lösung sein. Es liefert die Mitglieder einer Gruppe als Text zurück. So extrahieren Sie daraus die Benutzernamen:

```
function Get-LocalGroupMember($name = 'Administratoren') {
    $member = net localgroup $name
    $member[6..$($member.count-3)]
}
```

```
PS > Get-LocalGroupMember Benutzer
Nina
NT-AUTORITÄT\Authentifizierte Benutzer
NT-AUTORITÄT\INTERAKTIV
```

Lokale Benutzerkonten finden

Sie möchten wissen, welche lokalen Benutzerkonten es auf einem Computer gibt, oder Sie möchten ein bestimmtes Benutzerkonto finden, indem Sie nur einen Teil des Benutzernamens angeben.

Lösung

Der Befehl *net.exe* kann einfach und schnell alle lokalen Benutzerkonten auflisten:

```
PS > net user
```

```
Benutzerkonten für \\DEM05
```

```
-----
Administrator      Bitmarck      demo
Gast                Nina          PSTraining
testkonto           Tobias        Training
w7-pc9
Der Befehl wurde erfolgreich ausgeführt.
```

Das Ergebnis ist allerdings Text, und da die Benutzernamen mehrspaltig zurückgeliefert werden, lassen sich diese nur schlecht weiterverarbeiten.

Müssen die Benutzernamen weiterverarbeitet werden, verwenden Sie die folgende Funktion *Get-LocalUsers*. Ohne Parameterangaben liefert sie sämtliche lokalen Benutzerkonten:

```
Function Get-LocalUsers {  
    Param(  
        $username = '*',  
        $computername = $env:userdomain  
    )  
    $computer = [ADSI]"WinNT://$computername,computer"  
  
    $computer.psbase.children | Where-Object {$_.psbase.SchemaClassName -eq  
        'user'} | Where-Object { $_.Name -like $username }  
}
```

Mit dem Parameter *-username* können Sie auch nach Benutzernamen suchen, wobei Platzhalterzeichen erlaubt sind. Und mit dem Parameter *-computername* erfragen Sie lokale Benutzerkonten des Remotesystems. Die folgende Zeile liefert alle Benutzerkonten des Systems »storage1«, deren Name mit »a« beginnt:

```
PS > Get-LocalUsers -comp storage1 -user a*  
  
distinguishedName :  
Path              : WinNT://ARBEITSGRUPPE/storage1/Administrator  
  
distinguishedName :  
Path              : WinNT://ARBEITSGRUPPE/storage1/ASPNET
```

Hintergrund

Greifen Sie mit dem *WinNT:-ADsPath* auf ein Computersystem zu, kann es Benutzer, Gruppen und Dienste liefern, die im zugrunde liegenden *psbase*-Objekt in der Eigenschaft *Children* abrufbar sind. Der Typ eines zurückgelieferten Objekts steht in seiner *SchemaClassName*-Eigenschaft. Um also tatsächlich nur Benutzerkonten zu sehen, werden nur solche Objekte zurückgegeben, die vom Typ »user« sind. Möchten Sie lokale Gruppen auflisten, setzen Sie denselben Code ein und ändern im Filter »user« gegen »group« (siehe nächste Lösung).

Alle lokalen Gruppen auflisten

Sie möchten wissen, welche lokalen Benutzergruppen es auf einem Computer gibt, oder Sie möchten eine bestimmte Gruppe finden, indem Sie nur einen Teil des Namens angeben.

Lösung

Mit dem Befehl *net.exe* lassen sich alle lokalen Gruppen abrufen:


```
PS > net localgroup | Where-Object { $_.StartsWith('*') } | →
    ForEach-Object { $_.SubString(1) }
Administratoren
Benutzer
Distributed COM-Benutzer
Ereignisprotokollleser
Gäste
Hauptbenutzer
(...)
```

Oder verwenden Sie die folgende Funktion *Get-LocalGroups*. Ohne Parameterangaben liefert sie sämtliche lokalen Gruppen:

```
Function Get-LocalGroups {
    Param(
        $groupname = '*',
        $computername = $env:userdomain
    )
    $computer = [ADSI]"WinNT://$computername,computer"

    $computer.psbases.children | Where-Object { $_.psbase.SchemaClassName →
        -eq 'group' } | Where-Object { $_.Name -like $groupname }
}
```

Mit dem Parameter *-groupname* können Sie auch nach Gruppennamen suchen, wobei Platzhalterzeichen erlaubt sind. Und mit dem Parameter *-computername* erfragen Sie lokale Gruppen des Remotesystems. Die folgende Zeile liefert alle Gruppen des Systems »storage1«, deren Name das Wort »benutzer« enthält:

```
PS > Get-LocalGroups *benutzer* storage1

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Benutzer

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Distributed COM-Be...

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Hauptbenutzer

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Leistungsprotokoll...

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Remotedesktopbenutzer

distinguishedName :
Path              : WinNT://ARBEITSGRUPPE/storage1/Systemmonitorbenutzer
```

Hintergrund

Greifen Sie mit dem *WinNT*:-*ADsPath* auf ein Computersystem zu, kann es Benutzer, Gruppen und Dienste liefern, die im zugrunde liegenden *psbase*-Objekt in der Eigenschaft *Children* abrufbar sind. Der Typ eines zurückgelieferten Objekts steht in seiner *SchemaClassName*-Eigenschaft. Um also tatsächlich nur Gruppen zu sehen, werden nur solche Objekte zurückgegeben, die vom Typ *group* sind.

Prüfen, ob ein lokales Konto existiert

Sie möchten herausfinden, ob es ein bestimmtes lokales Benutzerkonto oder eine lokale Gruppe bereits gibt.

Lösung

Verwenden Sie die ADSI-Funktion *Exists()* und geben Sie den *ADsPath* des zu prüfenden Kontos an. Falls es schon existiert, wird *\$true* zurückgeliefert, andernfalls *\$false*:

```
PS > [ADSI]::Exists('WinNT://DEM05/Administratoren')
True
PS > [ADSI]::Exists('WinNT://DEM05/Administrator')
True
PS > [ADSI]::Exists('WinNT://DEM05/Administrator1')
False
```

Hintergrund

Die ADSI-Methode *Exists()* kann prüfen, ob es ein bestimmtes Konto bereits gibt. Allerdings liefert diese Funktion Fehlermeldungen zurück, wenn der *ADsPath* nicht korrekt angegeben wird. Deshalb kann man sie beispielsweise als Funktion *Test-LocalAccount* in einer eigenen PowerShell-Funktion kapseln und mit Fehlerhandling ausstatten:

```
Function Test-LocalAccount($ADsPath) {
    Try {
        [ADSI]::Exists($ADsPath)
    }
    catch { $false }
}

PS > Test-LocalAccount WinNT://DEM05/Administratoren
True
PS > Test-LocalAccount WinNT://DEM05/Administrator
True
PS > Test-LocalAccount WinNT://DEM05/Administrator1
False
PS > Test-LocalAccount WinNT://WORKGROUP/DEM05/Administrator1
False
PS > Test-LocalAccount WinNT://WORKGROUP/DEM05/Administrator
True
```

Neues lokales Benutzerkonto anlegen

Sie möchten ein neues lokales Benutzerkonto anlegen.

Lösung

Mithilfe des Befehls *net.exe* lassen sich sehr einfach neue lokale Konten einrichten. Die folgende Zeile legt ein Konto namens »KarlW« mit einem vordefinierten Kennwort an:

```
PS > net user KarlW topSecret99 /add
Der Befehl wurde erfolgreich ausgeführt.
```

Neue lokale Konten lassen sich programmgesteuert auch über das ADSI-Interface anlegen. Die Funktion *New-LocalUser* fasst die dabei notwendigen Schritte zusammen und generiert neue Benutzerkonten:

```
Function New-LocalUser {
    Param(
        [Parameter(Mandatory=$true)]
        $username,
        $password=$null,
        $description="",
        $fullname="",
        $computername = $env:userdomain
    )

    $computer = [ADSI]"WinNT://$computername,Computer"
    $user = $computer.Create("user", $username)
    If ($password) {
        $user.SetPassword($password)
    } else {
        $user.Put('PasswordExpired',1)
        $user.SetInfo()
    }
    $user.SetInfo()
    $user.Description = $description
    $user.FullName = $fullname
    $user.SetInfo()
    $user
}
```

So legen Sie ein neues lokales Benutzerkonto an und stattdessen es mit einem Kennwort aus:

```
PS > New-LocalUser -username KarlW -password P@ssw0rd -description 'von →
PowerShell angelegt' -fullname 'Karl Werner'
```

Geben Sie kein Kennwort an, wird das Konto so konfiguriert, dass sich der Anwender bei der nächsten Anmeldung selbst ein neues Kennwort auswählen muss. Mit Ausnahme des Parameters *-username* sind alle übrigen optional und können auch weggelassen werden. Geben Sie gar keine Parameter an, fragt die Funktion interaktiv nach dem Namen des neu zu erstellenden Kontos:

```
PS > New-LocalUser
```

```
Cmdlet New-LocalUser an der Befehlspipelineposition 1  
Geben Sie Werte für die folgenden Parameter an:  
username: karlw
```

Hintergrund

Um ein neues Benutzerkonto anzulegen, verbinden Sie sich zuerst via ADSI mit dem Container, in dem das neue Objekt gespeichert werden soll. Bei allen lokalen Benutzerkonten ist dies der lokale Computer. Danach wird mit der *Create()*-Methode des Containers das gewünschte Objekt angelegt. Dazu geben Sie den Typ des Objekts (*user*) und seinen Namen an.

Das Ergebnis ist ein *User*-Objekt, das sich bis jetzt allerdings nur im Speicher befindet. Ein echtes Benutzerkonto ist noch nicht angelegt worden. Als Nächstes legen Sie für das Konto mit *SetPassword()* ein initiales Kennwort fest, das den jeweiligen Kennwortrichtlinien entsprechen muss. Mit *SetInfo()* wird das *User*-Objekt nun tatsächlich angelegt. Die Kopie in Ihrem Computerspeicher wird erst jetzt in die SAM-Datenbank geschrieben. Das Benutzerkonto existiert nun.

Anschließend können Sie weitere Eigenschaften des neu angelegten Benutzerkontos festlegen, beispielsweise seine Beschreibung. Alle Änderungen am Objekt müssen zum Schluss einmalig mit *SetInfo()* erneut gespeichert werden. *SetInfo()* schließt also Änderungsvorgänge ab. Ohne *SetInfo()* werden keine Änderungen am Benutzerkonto vorgenommen.

Dies ist eine kleine Besonderheit von PowerShell. In herkömmlichen Programmier- und Skriptsprachen können Objekteigenschaften eines neuen Objekts auch ohne ein vorheriges *SetInfo()* gesetzt werden. Bei PowerShell ist dies erst möglich, wenn Sie mindestens einmal *SetInfo()* aufgerufen haben.

Dies kann zu einem Dilemma führen, wenn Sie Objekteigenschaften eines neuen Objekts setzen müssen, bevor Sie es mit *SetInfo()* instanziiieren können (sogenannte *Mandatory Properties*). In diesem Fall setzen Sie die Eigenschaft mit *Put()*. *Put()* kann auch auf vollkommen neue Objekte angewendet werden, ohne dass zuvor *SetInfo()* aufgerufen werden müsste:

```
PS > $computer = [ADSI]"WinNT://."  
PS > $user = $computer.Create("user", "Kurt")  
PS > $user.SetPassword("topsecret")  
PS > $user.Put('Description', 'Mit PowerShell angelegter neuer Benutzer')  
PS > $user.SetInfo()
```

TIPP

SetInfo() hat eine Entsprechung in .NET Framework. Sie können anstelle von *SetInfo()* auch *PSBase.CommitChanges()* aufrufen. Beide haben dieselbe Wirkung. Mit *PSBase* greifen Sie auf das zugrunde liegende .NET-Benutzerobjekt und sein eigenes Objektmodell zu.

Alle Benutzerkonten, die Sie auf diese Weise anlegen, erscheinen zunächst nicht in der Windows-Benutzerverwaltung, weil die Benutzerverwaltung nur solche Benutzerkonten anzeigt, die mindestens in der Gruppe *Benutzer* oder der Gruppe *Administratoren* Mitglied ist. Ihre neuen Benutzerkonten sind aber noch in keiner Gruppe Mitglied. Den Erfolg Ihres Codes können Sie aber über eine undokumentierte Erweiterung der Systemsteuerung dennoch begutachten. Geben Sie ein:

```
PS > control userpasswords2
```

Es öffnet sich ein Dialogfeld. Holen Sie darin die Registerkarte *Erweitert* in den Vordergrund und klicken Sie dann auf die Schaltfläche *Erweitert*. Klicken Sie anschließend in der Konsolenstruktur auf den Knoten *Benutzer*. Jetzt sehen Sie sämtliche lokale Benutzerkonten einschließlich der von Ihnen neu erstellten Konten.

Benutzerkonto in eine lokale Gruppe aufnehmen

Sie möchten ein Benutzerkonto zum Mitglied in einer lokalen Gruppe machen. Sie haben zum Beispiel ein neues Benutzerkonto mit *New-LocalUser* angelegt und möchten dieses Konto nun zum Mitglied der Gruppe der Administratoren machen.

Lösung

Am schnellsten fügen Sie ein Benutzerkonto mit dem Befehl *net.exe* in eine lokale Gruppe ein. Die folgende Zeile fügt das Konto »KarlW« in die lokale Gruppe »Administratoren« ein:

```
PS > net localgroup Administratoren /add KarlW  
Der Befehl wurde erfolgreich ausgeführt.
```

Programmgesteuert fügen Sie den *ADsPath* eines Kontos alternativ der Mitgliederliste einer Gruppe hinzu. Die folgende Funktion *Add-LocalGroup* erwartet zwei Parameter: die Gruppe, zu der hinzugefügt werden soll, und das Konto, das Sie hinzufügen wollen:

```
Function Add-LocalGroup($group, $newmember) {  
    $group.PSBase.Invoke("Add",($newmember | Select-Object -expand Path))  
}
```

So würden Sie mit den an anderer Stelle in diesem Kapitel vorgestellten Funktionen ein neues Benutzerkonto anlegen und danach der Gruppe der *Administratoren* hinzufügen:

```
PS > $admins = Get-LocalGroup Administratoren
PS > $newuser = New-LocalUser FHermann
PS > Add-LocalGroup -group $admins -new $newuser
```

Hintergrund

Jedes *Group*-Objekt enthält die Methode *Add()*, mit der unter Angabe eines gültigen *ADsPath* ein Mitglied der Gruppe hinzugefügt werden kann. Weil *Add()* nicht-öffentlich ist, muss die Methode über *Invoke()* aufgerufen werden.

Sie könnten ein Mitglied auch über eine einzelne PowerShell-Zeile zu einer Gruppe hinzufügen. Die folgende Zeile fügt den lokalen Benutzer »FrankT« in die Gruppe der *Administratoren* ein:

```
PS > ([ADSI]"WinNT://$env:computername/Administratoren").PSBase.Invoke ->
("Add","WinNT://$env:computername/FrankT")
```

Benutzerkonto aus einer lokalen Gruppe entfernen

Sie möchten ein Benutzerkonto aus einer lokalen Gruppe entfernen. Sie wollen zum Beispiel einem Konto Administratorechte entziehen und es deshalb aus der Gruppe der Administratoren entfernen.

Lösung

Am schnellsten wird ein Konto mit dem Befehl *net.exe* aus einer lokalen Gruppe entfernt. Die folgende Zeile entfernt das Konto »KarlW« aus der lokalen Gruppe »Administratoren«:

```
PS > net localgroup Administratoren /delete KarlW
Der Befehl wurde erfolgreich ausgeführt.
```

Programmgesteuert entfernen Sie alternativ den *ADsPath* eines Kontos aus der Mitgliederliste einer Gruppe. Die folgende Funktion *Remove-LocalGroup* erwartet zwei Parameter: die Gruppe, zu der hinzugefügt werden soll, und das Konto, das Sie hinzufügen wollen:

```
Function Remove-LocalGroup($group, $newmember) {
    $group.PSBase.Invoke("Remove",($newmember | >>
        Select-Object -expand Path))
}
```

So würden Sie mit den an anderer Stelle in diesem Kapitel vorgestellten Funktionen das Konto »FrankT« aus der Gruppe der Administratoren entfernen:

```
PS > $admins = Get-LocalGroup Administratoren
PS > $newuser = Get-LocalUser FrankT
PS > Remove-LocalGroup -group $admins -new $newuser
```

Hintergrund

Jedes *Group*-Objekt enthält die Methode *Remove()*, mit der unter Angabe eines gültigen *ADsPath* ein Mitglied aus der Gruppe entfernt werden kann. Weil *Remove()* nicht-öffentlich ist, muss die Methode über *Invoke()* aufgerufen werden.

Sie könnten ein Mitglied auch über eine einzelne PowerShell-Zeile aus einer Gruppe entfernen. Die folgende Zeile entfernt den lokalen Benutzer »FrankT« aus der Gruppe der Administratoren:

```
PS > ([ADSI]"WinNT://$env:computername/Administratoren").PSBase.Invoke ->
("Remove", "WinNT://$env:computername/FrankT")
```

Lokales Benutzerkonto konfigurieren

Sie möchten Einstellungen eines lokalen Benutzerkontos ändern. Sie wollen zum Beispiel dafür sorgen, dass ein Benutzer bei seiner nächsten Anmeldung ein neues Kennwort wählen muss, oder Sie möchten das Kennwort eines Kontos ändern.

Lösung

Die Einstellungen werden im Benutzerobjekt direkt hinterlegt. Dazu beschaffen Sie sich zuerst das gewünschte Benutzerkonto, zum Beispiel über die in diesem Kapitel beschriebene Funktion *Get-LocalUser*:

```
PS > $user = Get-LocalUser Testuser
```

Möchten Sie, dass der Kontoinhaber bei der nächsten Anmeldung sein Kennwort ändern muss, ändern Sie die Eigenschaft *PasswordExpired* und erklären damit das aktuelle Kennwort für abgelaufen:

```
PS > $user.Put('PasswordExpired',1)
PS > $user.SetInfo()
```

Andere Einstellungen sind als Bitmaske in der Eigenschaft *UserFlags* verborgen. Möchten Sie, dass das Kennwort eines Kontos niemals abläuft, setzen Sie das entsprechende Bit darin:

```
PS > $passwordnotrequired = 0x20
PS > $passwordcantchange = 0x40
PS > $neverexpires = 0x10000
PS > $user = Get-LocalUser Testing
PS > $user.userflags = $user.userflags.value -bor $neverexpires
PS > $user.SetInfo()
```

ACHTUNG Alle Änderungen an den Eigenschaften werden erst dann wirksam, wenn Sie *SetInfo()* aufrufen. Dies braucht nur einmal nach Abschluss aller Änderungen zu geschehen:

```
PS > $user.SetInfo()
```

Wollen Sie das Kennwort des Kontos neu festsetzen, setzen Sie *SetPassword()* ein:

```
PS > $user.SetPassword('NewP@ssw0rd')
```

Hintergrund

Die Konfiguration eines Benutzerkontos erfolgt über seine verschiedenen Eigenschaften, was je nachdem, welche Einstellung des Kontos Sie ändern wollen, sehr einfach oder auch recht komplex sein kann. Wollen Sie beispielsweise nur die Textbeschreibung des Kontos ändern, weisen Sie der Eigenschaft *Description* eine neue Beschreibung zu und rufen *SetInfo()* auf, um die Änderungen wirksam werden zu lassen. Möchten Sie dagegen das Konto so konfigurieren, dass Kennwörter nie ablaufen, muss dazu das passende Bit in der Eigenschaft *UserFlags* gesetzt werden, was schon höheren Aufwand nötig macht.

Das Kennwort des Kontos kann überhaupt nicht mittels Eigenschaften verändert werden. Hierfür sind die Methoden *SetPassword()* und *ChangePassword()* nötig. Setzen Sie das Kennwort eines Kontos mit *SetPassword()* neu, benötigen Sie Administratorrechte und das Konto verliert unter Umständen den Zugriff auf EFS-verschlüsselte Daten im Dateisystem. Ist das alte Kennwort dagegen bekannt, kann ein neues Kennwort ohne Administratorrechte und ohne Folgen für EFS-Zertifikate geändert werden:

```
PS > $user.ChangePassword('altesKennwort', 'neuesKennwort')
```

Prüfen, ob ein lokales Benutzerkonto Mitglied einer Gruppe ist

Sie möchten wissen, ob ein Benutzerkonto Mitglied einer bestimmten Gruppe ist.

Lösung

Um zu prüfen, ob ein Benutzerkonto Mitglied in einer bestimmten Gruppe ist, greifen Sie per ADSI auf die Gruppe zu und testen dann mit *isMember()*, ob ein bestimmter Benutzer darin Mitglied ist. Dazu geben Sie für den Benutzer den *ADsPath* seines Benutzerkontos an.

TIPP Den *ADsPath* eines Benutzers finden Sie beispielsweise mit der Funktion *Get-LocalUser* heraus, die in diesem Kapitel beschrieben wurde. Den *ADsPath* des aktuellen Benutzers lesen Sie beispielsweise so:


```
PS > Get-LocalUser | Select-Object -expandProperty Path
WinNT://WORKGROUP/DEM05/w7-pc9
```

Die folgenden Zeilen prüfen, ob der aktuelle Benutzer (direktes) Mitglied in der Gruppe der Administratoren ist:

```
PS > $computername = $env:computername
PS > $gruppenname = 'Administratoren'
PS > $username = $env:username
PS > $userdomain = $env:userdomain
PS > $group = [ADSI]"WinNT://$computername/$gruppenname,group"
PS > if ($userdomain -eq $computername) { $userdomain = "workgroup/$userdomain" }
PS > $group.isMember("WinNT://$userdomain/$username")
True
```

Hintergrund

Jede Gruppe verfügt über eine Methode namens *isMember()*. Gibt man dieser einen *ADsPath* an, meldet sie, ob dieser Mitglied der Gruppe ist.

Das funktioniert zwar einwandfrei, aber bei der Angabe des *ADsPath* kann es insbesondere bei Computern zu Rätselraten kommen, die kein Mitglied in einer Domäne sind. In diesem Fall nämlich verlangt *isMember()* den Arbeitsgruppennamen

Möchten Sie wissen, wer Mitglied in einer lokalen Gruppe ist, rufen Sie die Methode *Members()* des *Group*-Objekts auf und geben dann für jedes Mitglied dessen *ADsPath* aus. Weil diese Methoden und Eigenschaften nicht-öffentlich sind, müssen sie auf etwas verschlungenem Wege angesprochen werden. Die folgenden Zeilen liefern alle Mitglieder der lokalen Gruppe *Administratoren*:

```
PS > $computername = $env:computername
PS > $gruppenname = 'Administratoren'
PS > $group = [ADSI]"WinNT://$computername/$gruppenname,group"
PS > $group.psbase.invoke("Members") | ForEach-Object {$_.GetType() →
    .InvokeMember('ADsPath', 'GetProperty', $null, $_, $null)}
WinNT://WORKGROUP/DEM05/Administrator
WinNT://WORKGROUP/DEM05/demo
WinNT://WORKGROUP/DEM05/Tobias
WinNT://WORKGROUP/DEM05/Training
WinNT://WORKGROUP/DEM05/Testing
WinNT://WORKGROUP/DEM05/RemoteTestuser
```

Lokales Benutzerkonto aktivieren oder deaktivieren

Sie möchten ein lokales Benutzerkonto vorübergehend deaktivieren, zum Beispiel, weil der Benutzer längere Zeit abwesend sein wird. Oder Sie möchten ein deaktiviertes Konto reaktivieren oder entsperren.

Lösung

Greifen Sie mit der in diesem Kapitel vorgestellten Funktion *Get-LocalUser* auf das Konto zu und ändern Sie die Eigenschaft *AccountDisabled* auf *\$true*.

```
PS > $user = Get-LocalUser KarlW
PS > $user.PSBase.InvokeSet('AccountDisabled',$true)
PS > $user.SetInfo()
```

Um ein Konto wieder zu reaktivieren, setzen Sie die Eigenschaft auf *\$false*:

```
PS > $user.PSBase.InvokeSet('AccountDisabled',$false)
PS > $user.SetInfo()
```

Wurde das Konto vom System gesperrt, zum Beispiel, weil sich der Benutzer zu häufig mit einem falschen Kennwort angemeldet hat, entsperren Sie es über die Eigenschaft *isAccountLocked*:

```
PS > $user.isAccountLocked = $false
PS > $user.SetInfo()
```

Hintergrund

Ob ein Konto verwendet werden darf oder nicht, regelt die Eigenschaft *AccountDisabled*. Diese Eigenschaft ist im Benutzerobjekt, das PowerShell liefert, allerdings nicht enthalten. Über *PSBase* greift man deshalb auf das zugrunde liegende »rohe« Benutzerobjekt zurück, wo die Eigenschaft zu finden ist.

Anders verhält es sich bei gesperrten Konten. Die Sperrung wird über die Eigenschaft *isAccountLocked* geregelt. Sie kann nur auf *\$false* eingestellt werden, um das Konto zu entsperren, wofür Administratorrechte nötig sind. Sperren kann man ein Konto dagegen nicht. Dies kann nur das Betriebssystem selbst.

Damit die Änderungen an Eigenschaften eines Benutzerobjekts wirksam werden, muss anschließend *SetInfo()* aufgerufen werden.

Lokales Benutzerkonto löschen

Sie möchten ein vorhandenes lokales Benutzerkonto löschen.

Lösung

Das Löschen eines Benutzerkontos ist ein kritischer und irreversibler Vorgang, den Sie nicht unbedacht ausführen dürfen.

Am schnellsten wird ein lokales Benutzerkonto mit dem Befehl *net.exe* entfernt. Die folgende Zeile entfernt das lokale Benutzerkonto »KarlW«:

```
PS > net user KarlW /delete
Der Befehl wurde erfolgreich ausgeführt.
```

Programmgesteuert entfernen Sie lokale Benutzerkonten mit einer Funktion wie *Remove-Local-User*:

```
Function Remove-LocalUser {
    Param(
        $username = $env:username,
        $computername = $env:userdomain
    )
    $computer = [ADSI]"WinNT://$computername,computer"
    $computer.Delete('user', $username)
}
```

Die folgende Zeile löscht den lokalen Benutzer »KarlW«:

```
PS > Remove-LocalUser KarlW
```

Verbinden Sie sich per *[ADSI]* und dem *WinNT*:-Provider mit dem Computer, auf dem Sie ein lokales Benutzerkonto löschen wollen. Möchten Sie das Konto auf dem eigenen Computer löschen, verwenden Sie anstelle des Rechnernamens einen Punkt. Löschen Sie dann mit *Delete()* das Benutzerkonto.

ACHTUNG Sie löschen damit das angegebene Benutzerkonto unwiderruflich. Sie können ein gelöscht Benutzerkonto nicht ohne Weiteres wiederherstellen, weil jedes neue Konto eine eigene unverwechselbare Sicherheits-ID erhält. Haben Sie also versehentlich ein Konto gelöscht und es danach neu angelegt, fehlen dem Konto nach wie vor sämtliche Berechtigungen und Gruppenmitgliedschaften des ursprünglichen Benutzerkontos. Seien Sie deshalb besonders vorsichtig, wenn Sie Benutzerkonten löschen.

```
PS > $computer = [ADSI]"WinNT://."
PS > $computer.Delete("user", "Karlheinz")
```

Hintergrund

Genauso, wie Sie mit *Create()* neue Objekte in einen ADSI-Container hineinlegen können, lassen sich mit *Delete()* Objekte auch daraus entfernen. Ein Aufruf von *SetInfo()* ist wie bei allen Methoden nicht erforderlich. Das angegebene Benutzerkonto wird sofort unwiderruflich gelöscht. *SetInfo()* benötigen Sie nur, wenn Sie Eigenschaften des Objekts geändert haben.

Zusammenfassung

Dem Modul *ActiveDirectory* mit den zahlreichen speziellen Active Directory-Cmdlets gehört die Zukunft. Die vielfältigen Lösungen in diesem Kapitel haben gezeigt, dass man diese zwar auch mit .NET-Bordmitteln lösen kann, doch ist die Arbeit mit den Cmdlets schneller, zuverlässiger und intuitiver.

Die Frage ist höchstens, wann diese Zukunft auch für Ihr Active Directory anbricht. Sie benötigen mindestens einen (kostenfreien) Gatewaydienst, um über die neuartigen Webservices kommunizieren zu können. Noch besser (aus Sicht von Microsoft) ist natürlich ein Rechner mit Windows Server 2008 R2 als Domänencontroller, denn der kommt bereits mit Gateway und dem Modul *ActiveDirectory*.

Aber auch ohne Windows Server 2008 R2 lohnt es sich, den relativ geringen Aufwand zu treiben, damit man die neuartigen Active Directory-Cmdlets einsetzen kann. Spätestens wenn Sie mehr als nur gelegentlich in Active Directory zu tun haben, sparen die Cmdlets enorm viel Zeit.

Alternativ könnten Sie auch zu den bewährten (und ebenfalls kostenfreien) Active Directory-Cmdlets der Firma Quest greifen. Sie wurden zwar nicht in diesem Kapitel eingesetzt, funktionieren aber ganz ähnlich. Im Gegensatz zu den Microsoft-Cmdlets benötigen sie keinen Gatewaydienst, weil sie noch auf klassischem Wege per DCOM kommunizieren.

Spätestens wenn es an die Verwaltung lokaler Konten geht, helfen weder die einen noch die anderen. Hier sind Low-Level-.NET-Methoden gefragt und deshalb haben Sie für alle wichtigen Verwaltungsaufgaben im Bereich lokaler Benutzerkonten entsprechende Beispiele und Funktionen gefunden. Diese sind zwar noch nicht mit Hilfe und Fehlerhandling ausgestattet, um den relevanten Code klein zu halten, bieten aber sicher eine gute Grundlage, um eigene Aufgaben umzusetzen.

Alternativ steht für viele Aufgaben rund um lokale Benutzerkonten und -gruppen das Befehlszeilentool *net.exe* zur Verfügung. Es lässt sich schnell und einfach einsetzen, kann aber nur bedingt in die Abläufe eines Skripts integriert werden.

Erstens liefert es Informationen als reinen Text zurück, der noch dazu neben den gewünschten Informationen Beschreibungen und Statusmeldungen enthält, die es herauszufiltern gilt. Und zweitens lassen sich Fehler, die durch *net.exe* ausgelöst werden, nicht im Rahmen der normalen .NET-Fehlerbehandlung abfangen (*trap* oder *try/catch*). Stattdessen müssen Fehler – wie bei Konsolenbefehlen üblich – manuell abgefangen werden, indem der Rückgabewert des Konsolenbefehls aus *\$LASTEXITCODE* ausgewertet wird:

```
PS > function Remove-User($name) { net user $name /delete 2>&1 | →  
    Out-Null; $LASTEXITCODE }  
PS > Remove-User KarlW  
0  
PS > Remove-User KarlW  
2
```

Kapitel 17

PowerShell-Snap-Ins und Module

In diesem Kapitel:

Feststellen, welche Erweiterungen verfügbar sind	494
PowerShell-Erweiterungen laden	496
Neue Cmdlets einer Erweiterung sichtbar machen	497
Neue Provider einer Erweiterung sichtbar machen	499
Eigene PowerShell-Erweiterungen herstellen	502
Zusammenfassung	505

PowerShell ist erweiterbar, und wenn Sie neue Cmdlets oder Provider zur Verwaltung von Active Directory, von Exchange oder VMware benötigen, rüsten Sie diese einfach nach.

PowerShell ist nämlich modular aufgebaut, und die 236 Cmdlets (Befehle) und acht Provider (stellen virtuelle Laufwerke bereit), die im Lieferumfang von PowerShell enthalten sind, stammen aus insgesamt sieben automatisch geladenen Erweiterungen, die nur als Grundausstattung dienen.

Dabei stehen zwei Typen von Erweiterungen zur Verfügung:

- **PSSnapin** Diese Erweiterung muss stets mit Administrator-Rechten installiert werden, weil jedes PSSnapin sich im geschützten Zweig *HKEY_LOCAL_MACHINE* der Registrierungsdatenbank registriert. PSSnapins gibt es seit PowerShell Version 1.0. Alle sieben standardmäßig geladenen Erweiterungen sind PSSnapins.
- **Modul** Module müssen nicht installiert werden. Es genügt, den Pfad zum Modul anzugeben. Damit können Module per Copy & Paste-Deployment verteilt werden. Module können zudem mit einfachen Mitteln selbst erstellt werden. Module wurden mit PowerShell Version 2.0 eingeführt. In Kapitel 16 haben Sie das Modul *ActiveDirectory* der Firma Microsoft eingesetzt, um 76 neue Cmdlets und einen neuen Provider zur Verwaltung von Active Directory nachzurüsten.

In diesem Kapitel erfahren Sie, wie Sie herausfinden, welche zusätzlichen Erweiterungen auf Ihrem System zur Verfügung stehen und wie Sie sie einsetzen.

Feststellen, welche Erweiterungen verfügbar sind

Sie möchten wissen, welche zusätzlichen PowerShell-Erweiterungen auf Ihrem System zur Verfügung stehen.

Lösung

Um alle verfügbaren PSSnapins zu sehen, rufen Sie *Get-PSSnapin* mit dem Parameter *-Registered* auf:

```
PS > Get-PSSnapin -Registered
```

```
Name       : Pscx
PSVersion  : 2.0
Description : PowerShell Community Extensions (PSCX) base snapin which
              implements a general purpose set of cmdlets.
```

```
Name       : Quest.ActiveRoles.ADManagement
PSVersion  : 1.0
Description : This Windows PowerShell snap-in contains cmdlets to manage
              Active Directory and Quest ActiveRoles Server.
```

```
Name       : VMware.VimAutomation.Core
PSVersion  : 2.0
Description : This Windows PowerShell snap-in contains Windows
              PowerShell cmdlets used to manage VMware Infrastructure.
```

Erhalten Sie keine Resultate, dann sind zurzeit keine weiteren PSSnapins auf Ihrem System installiert.

Um verfügbare Module zu sehen, rufen Sie *Get-Module* mit dem Parameter *-ListAvailable* auf:

```
PS > Get-Module -ListAvailable
```

ModuleType	Name	ExportedCommands
-----	----	-----
Manifest	ActiveDirectory	{}
Manifest	AppLocker	{}
Manifest	BitsTransfer	{}
Manifest	GroupPolicy	{}
Manifest	PSDiagnostics	{}
Manifest	TroubleshootingPack	{}

Dieser Befehl sollte immer Resultate liefern, weil PowerShell 2.0 das Modul *BitsTransfer* mitliefert.

Hintergrund

Da sich Module nicht offiziell registrieren (und deshalb der Parameter von *Get-Module* auch *-ListAvailable* und nicht *-Registered* heißt), kann *Get-Module -ListAvailable* nur solche Module anzeigen, die sich »inoffiziell« registriert haben. Dies geschieht, indem das Modul an einem von zwei Orten im Dateisystem abgelegt wird. Hier schaut *Get-Module* nach:

- **Benutzerspezifische Module** Sie stehen nur dem aktuellen Benutzer zur Verfügung und liegen in seinem Benutzerprofil: *dir "\$(Split-Path \$profile)\Modules"*
- **Allgemeine Module** Sie stehen allen Benutzern zur Verfügung und liegen im Windows-Ordner: *dir "\$(Split-Path \$profile.AllUsersAllHosts)\Modules"*

Befindet sich das Modul an einem anderen Ort, zum Beispiel auf einem freigegebenen Netzlaufwerk oder auf einem USB-Stick, wird das Modul nicht aufgelistet und muss bei Bedarf über seinen Pfadnamen geladen werden.

PSSnapins müssen sich im Gegensatz zu Modulen bei der Installation in der Registrierungsdatenbank registrieren. Deshalb findet *Get-PSSnapin -Registered* stets alle verfügbaren PSSnapins.

HINWEIS

»Verfügbar« ist dabei wörtlich gemeint. Da es PSSnapins geben kann, die nur in einer 32-Bit-Umgebung oder nur in einer 64-Bit-Umgebung funktionieren, zeigt *Get-PSSnapin* jeweils nur die PSSnapins an, die in der aktuellen Umgebung geladen werden können. Auf 64-Bit-Systemen stehen Ihnen deshalb zwei PowerShell-Konsolen zur Verfügung. Neben der Standard-PowerShell, die hier in 64 Bit läuft, gibt es noch *Windows PowerShell (x86)*, die als 32-Bit-Prozess läuft und gegebenenfalls andere PSSnapins anzeigt als die 64-Bit-Umgebung.

Bei der Registrierung werden die PSSnapins im Zweig *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\PowerShellSnapIns* in der Windows-Registrierungsdatenbank hinterlegt. Eine Liste sämtlicher installierter Snap-Ins erhalten Sie also zum Beispiel auch so:

```
PS > dir HKLM:\Software\Microsoft\PowerShell\1\PowerShellSnapIns
```

PSSnapins und Module, die Ihnen aufgelistet werden, sind nur verfügbar, aber noch nicht einsatzbereit. Möchten Sie die Cmdlets und/oder Provider daraus nutzen, müssen die Erweiterungen zuerst mit *Add-PSSnapin* bzw. *Import-Module* geladen werden.

PowerShell-Erweiterungen laden

Sie möchten eine Erweiterung nachladen, um neue Cmdlets und/oder Provider nutzen zu können. Sie wollen zum Beispiel eine Datei aus dem Internet herunterladen und dazu die Cmdlets des Moduls *BitsTransfer* nutzen. Oder Sie haben vielleicht ein PSSnapin eines Drittanbieters wie VMware installiert und möchten damit virtuelle Maschinen konfigurieren.

Lösung

Handelt es sich bei der Erweiterung um ein Modul, setzen Sie *Import-Module* ein und geben den Namen des Moduls an:

```
PS > Import-Module BitsTransfer
```

Der Name des Moduls darf mit Platzhalterzeichen abgekürzt werden und Sie können auch sämtliche verfügbaren Module in einem Schritt laden:

```
PS > Import-Module Bits*  
PS > Import-Module *
```

Ist das Modul nicht an einem der Standardorte für Module gespeichert, geben Sie anstelle des Modulnamens den Pfadnamen zum Modulordner an:

```
PS > Import-Module k:\scripts\unit1\modules\networktools
```

Handelt es sich bei der Erweiterung um ein PSSnapin, setzen Sie *Add-PSSnapin* ein und geben den Namen des PSSnapin an.

```
PS > Add-PSSnapin VMware.VimAutomation.Core  
PS > Add-PSSnapin VMWare*
```

Hintergrund

PSSnapins und Module werden pro Sitzung geladen. Sie müssen die Erweiterungen, die Sie nutzen möchten, also bei jedem Start der PowerShell neu auswählen und nachladen. Arbeiten Sie häufig mit einer Erweiterung, sollten Sie sie deshalb aus Ihrem Profilskript heraus automatisch

nachladen lassen. Der Ort Ihres Profilskripts findet sich in der Variablen *\$profile*. Es kann sein, dass das Profilskript bei Ihnen noch nicht existiert und erst noch angelegt werden muss.

Das mehrfache Laden eines Moduls ist erlaubt und führt zu keiner Fehlermeldung. Allerdings lädt PowerShell ein Modul, das bereits importiert worden war, kein zweites Mal, sondern ignoriert dann einfach den Importbefehl. In Produktionsumgebungen ist das sinnvoll. In Entwicklungsumgebungen, bei denen Module erstellt und revidiert werden, laden Sie ein geändertes Modul neu, indem Sie den Parameter *-Force* angeben oder das Modul mit *Remove-Module* zuerst wieder entfernen.

Das mehrfache Laden eines PSSnapins ist nicht erlaubt und führt zu einer Fehlermeldung. Sie können diese Fehlermeldung mit *-ErrorAction SilentlyContinue* aber unterdrücken lassen.

Module können Skripts enthalten und lassen sich in diesem Fall nur dann korrekt importieren, wenn Sie mit der ExecutionPolicy (Ausführungsrichtlinie) die Ausführung von Skripts erlaubt haben:

```
PS > Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Stammt das Modul aus einem Netzlaufwerk oder von einem anderen Ort, der nicht Teil Ihrer Domäne ist, kann es sein, dass bei jedem Import eine Sicherheitsabfrage erscheint oder das Modul gar nicht geladen werden kann. Schuld ist in diesem Fall die ExecutionPolicy, die den Speicherort für »remote« hält und deshalb eine gültige digitale Signatur fordert. Sie lösen das Problem, indem Sie die Signaturprüfung für die aktuelle Sitzung oder dauerhaft abschalten:

```
PS > Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass  
PS > Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Bypass
```

Beim Import eines Moduls kann es außerdem zu einer (harmlosen) gelben Warnung kommen:

```
WARNUNG: Einige importierte Befehle enthalten nicht genehmigte Verben. Dies kann deren Auffindbarkeit erschweren. Weitere Informationen erhalten Sie mithilfe des Verbose-Parameters, oder geben Sie "Get-Verb" ein, um eine Liste der genehmigten Verben anzuzeigen.
```

In diesem Fall hat der Autor des Moduls für seine Cmdlet-Namen Verben verwendet, die nicht in der offiziell zugelassenen Verbenliste stehen. Diese Liste können Sie mit *Get-Verb* abrufen. Das Modul funktioniert trotz dieser Warnung einwandfrei.

Neue Cmdlets einer Erweiterung sichtbar machen

Sie haben eine PowerShell-Erweiterung nachgeladen und möchten nun wissen, welche neuen Cmdlets Ihnen zur Verfügung stehen.

Lösung

Rufen Sie die Cmdlets mit *Get-Command* ab und schränken Sie die Liste mit *-Module* auf die Cmdlets ein, die aus einer bestimmten Erweiterung stammen.

```
PS > Import-Module BitsTransfer
PS > Get-Command -Module BitsTransfer
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Add-BitsFile	Add-BitsFile [-BitsJob] <Bit...
Cmdlet	Complete-BitsTransfer	Complete-BitsTransfer [-Bits...
Cmdlet	Get-BitsTransfer	Get-BitsTransfer [[-Name] <S...
Cmdlet	Remove-BitsTransfer	Remove-BitsTransfer [-BitsJo...
Cmdlet	Resume-BitsTransfer	Resume-BitsTransfer [-BitsJo...
Cmdlet	Set-BitsTransfer	Set-BitsTransfer [-BitsJob] ...
Cmdlet	Start-BitsTransfer	Start-BitsTransfer [-Source]...
Cmdlet	Suspend-BitsTransfer	Suspend-BitsTransfer [-BitsJ...

```
PS > Add-PSSnapin VMWare*
PS > Get-Command -Module VMWare*
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Add-VMHost	Add-VMHost [-Name] <String>...
Cmdlet	Add-VMHostNtpServer	Add-VMHostNtpServer [-NtpSer...
Cmdlet	Connect-VIServer	Connect-VIServer [-Server] <...
Cmdlet	Disconnect-VIServer	Disconnect-VIServer [[-Serve...
Cmdlet	Dismount-Tools	Dismount-Tools [[-Guest] <VM...

Hintergrund

Häufig ist der erste Schritt nach dem Laden einer Erweiterung, sich einen Überblick über die darin enthaltenen Befehle zu verschaffen. *Get-Command* unterstützt mit *-Module* eine praktische Filterung, die nur diejenigen Cmdlets anzeigt, die aus einer bestimmten Erweiterung stammen. Dabei ist es gleichgültig, ob es sich bei der Erweiterung um ein Modul oder ein PSSnapin handelt. Deshalb unterstützt der Parameter *-Module* auch den Aliasnamen *-PSSnapin* und beide Parameternamen sind gegeneinander austauschbar.

Interessieren Sie sich anschließend näher für eines der neuen Cmdlets und möchten Sie vielleicht wissen, wie Sie mit *Start-BitsTransfer* eine Datei aus dem Internet herunterladen, rufen Sie die Hilfe – oder noch besser – die Beispiele für den Befehl ab:

```
PS > help Start-BitsTransfer -examples
```

NAME

Start-BitsTransfer

ÜBERSICHT

Erstellt einen neuen Übertragungsauftrag des intelligenten Hintergrundübertragungsdiensts (BITS).

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

```
C:\PS>Start-BitsTransfer -Source
http://server01/servertestdir/testfile1.txt
-Destination c:\clienttestdir\testfile1.txt
```

Beschreibung

Mit diesem Befehl erstellen Sie einen neuen BITS-Übertragungsauftrag, mit dem eine Datei von einem Server heruntergeladen wird. Der lokale und der Remotename der Datei werden im Source-Parameter und im Destination-Parameter angegeben. Da der Standardübertragungstyp "Download" ist, wird die Datei "http://Server01/servertestdir/testfile1.txt" auf den Client nach "C:\clienttestdir\testfile1.txt" übertragen. Die Eingabeaufforderung kehrt zurück, sobald die Dateiübertragung abgeschlossen wurde oder ein Fehler aufgetreten ist.

-- Fortsetzung --

Neue Provider einer Erweiterung sichtbar machen

Sie möchten wissen, ob eine nachgeladene Erweiterung neue Provider enthält, und falls ja, möchten Sie mit dem neuen Provider ein virtuelles Laufwerk anlegen.

Lösung

Listen Sie die Provider mit *Get-PSProvider* auf und beschränken Sie die Liste auf alle Provider aus einer bestimmten Erweiterung. Die folgenden Zeilen listen alle Provider des Moduls *ActiveDirectory* auf (das wie in Kapitel 16 beschrieben aus den RSAT-Tools nachgerüstet werden kann):

```
PS > Import-Module ActiveDirectory
```

```
PS > PS > Get-PSProvider | Where-Object { $_.Module -like '*Active*' }
```

Name	Capabilities	Drives
----	-----	-----
ActiveDirectory	Include, Exclude, Filter, ... {}	

Um zu sehen, ob dieser Provider bereits virtuelle Laufwerke bereitstellt, lassen Sie mit *Get-PSDrive* alle virtuellen Laufwerke auflisten und beschränken diese auf solche, die den ermittelten Provider einsetzen:

```
PS > Get-PSDrive -PSProvider ActiveDirectory
```

Möchten Sie ein neues Laufwerk mit dem Provider anlegen, verwenden Sie *New-PSDrive*. Im einfachsten Fall geben Sie Name, Providernamen und Wurzelverzeichnis des neuen Laufwerks an:

```
PS > New-PSDrive -name AD1 -psprovider ActiveDirectory -root ''
```

Sind Anmeldeinformationen erforderlich, können diese mit *-Server* und *-Credential* angegeben werden:

```
PS > New-PSDrive -name AD1 -psprovider ActiveDirectory -root '' -Server 192.168.2.234 -Credential powershell\Tobias
```

WARNUNG: Die Spalte CurrentLocation passt nicht in die Anzeige und wurde entfernt.

Name	Used (GB)	Free (GB)	Provider	Root
----	-----	-----	-----	----
AD1			ActiveDire...	//RootDSE/

Sie können nun mithilfe des gewählten Laufwerksbuchstaben auf das Laufwerk zugreifen (hängen Sie wie bei allen Laufwerksbuchstaben ein »:« an den Namen an):

```
PS > dir ad1:
```

Name	ObjectClass	DistinguishedName
----	-----	-----
powershell	domainDNS	DC=powershell,DC=local
Configuration	configuration	CN=Configuration,DC=powershell,DC=...
Schema	dMD	CN=Schema,CN=Configuration,DC=powe...
DomainDnsZones	domainDNS	DC=DomainDnsZones,DC=powershell,DC...
ForestDnsZones	domainDNS	DC=ForestDnsZones,DC=powershell,DC...

Hintergrund

Provider enthalten das »Wissen«, um hierarchische Informationsspeicher wie Laufwerke anzeigen zu lassen. PowerShell kommt bereits mit einer Reihe von Providern, mit denen Sie beispielsweise die Registrierungsdatenbank verwalten:

```
PS > Dir HKLM:\Software
```

Erweiterungen können weitere Provider liefern. Das Modul *ActiveDirectory* rüstet zum Beispiel einen gleichnamigen Provider nach, mit dem sich Active Directory wie ein Laufwerk anzeigen und verwalten lässt. Im Falle des *ActiveDirectory*-Providers würde das Laufwerk als Vorgabe zur Navigation allerdings *Distinguished Names* (definierte Namen) erfordern, was etwas lästig ist:

```
PS > cd 'ad1:DC=powershell,DC=local'
PS > dir
```

Name	ObjectClass	DistinguishedName
Builtin	builtinDomain	CN=Builtin,DC=powershell,DC=local
Computers	container	CN=Computers,DC=powershell,DC=l...
Domain Controllers	organizationalUnit	OU=Domain Controllers,DC=powers...
ForeignSecurity...	container	CN=ForeignSecurityPrincipals,DC...
Infrastructure	infrastructureU...	CN=Infrastructure,DC=powershell...
(...)		
Users	container	CN=Users,DC=powershell,DC=local
Vertrieb	organizationalUnit	OU=Vertrieb,DC=powershell,DC=local

Geben Sie mit *-formatType Canonical* beim Erstellen des Laufwerks das kanonische Format an, erlaubt das Laufwerk eine wesentlich intuitivere Navigation mithilfe der kurzen kanonischen Namen:

```
PS > cd c:\
PS > Remove-PSDrive AD1
PS > New-PSDrive -name AD1 -psprovider ActiveDirectory -root '' -Server →
192.168.2.234 -Credential powershell\Tobias -formatType Canonical
```

Name	Used (GB)	Free (GB)	Provider	Root
AD1			ActiveDire...	//RootDSE/

```
PS > dir
```

Name	ObjectClass	DistinguishedName	CanonicalName
powershell	domainDNS	DC=powershell,D...	powershell.local/
Configuration	configuration	CN=Configuratio...	powershell.loc...
Schema	dMD	CN=Schema,CN=Co...	powershell.loc...
DomainDnsZones	domainDNS	DC=DomainDnsZon...	DomainDnsZones...
ForestDnsZones	domainDNS	DC=ForestDnsZon...	ForestDnsZones...

```
PS > cd powershell.local/users
PS > dir
```

Name	ObjectClass	DistinguishedName	CanonicalName
Abgelehnte RODC-K...	group	CN=Abgelehnte R...	powershell.loca...
Administrator	user	CN=Administrato...	powershell.loca...
DnsAdmins	group	CN=DnsAdmins,CN...	powershell.loca...
DnsUpdateProxy	group	CN=DnsUpdatePro...	powershell.loca...
Domänen-Admins	group	CN=Domänen-Admi...	powershell.loca...
Domänen-Benutzer	group	CN=Domänen-Benu...	powershell.loca...
Domänen-Gäste	group	CN=Domänen-Gäst...	powershell.loca...

TIPP

Der *ActiveDirectory*-Provider erwartet in Pfadnamen den normalen Schrägstrich »/« und nicht den Backslash »\«, der zu einer Fehlermeldung führt.

Eigene PowerShell-Erweiterungen herstellen

Sie möchten ein eigenes Erweiterungsmodul für PowerShell schreiben, um beispielsweise einen Satz nützlicher Funktionen als Modul an Kollegen weiterzugeben.

Lösung

Verfassen Sie zuerst ein Skript, das die Funktionen bereitstellt, die das Modul künftig bereitstellen soll. Das folgende Skript definiert beispielsweise die Funktion *Get-SoftwareUpdates* und liefert eine Liste der Software-Updates, die auf dem Computer installiert wurden:

```
function Get-SoftwareUpdates {  
    $filter = @{  
        logname='Microsoft-Windows-Application-Experience/Program-Inventory'  
        id=905  
    }  
  
    Get-WinEvent -FilterHashtable $filter |  
    ForEach-Object {  
        $info = 1 | Select-Object Datum, Anwendung, Version, Herausgeber  
        $info.Datum = $_.TimeCreated  
        $info.Anwendung = $_.Properties[0].Value  
        $info.Version = $_.Properties[1].Value  
        $info.Herausgeber = $_.Properties[2].Value  
        $info  
    }  
}
```

Legen Sie sich als Nächstes einen Modulordner an. Der Ordnername entspricht dem Namen, unter dem das Modul später angesprochen wird. Speichern Sie Ihr Skript in diesem Ordner unter einem beliebigen Namen.

Legen Sie nun in demselben Ordner die folgende Datei an, die genauso heißen muss wie der Ordner selbst und die die Erweiterung *.psm1* tragen muss:

```
. $PSScriptRoot\skriptname.ps1
```

Ersetzen Sie *skriptname.ps1* durch den Namen des Skripts, das Sie im Modulordner gespeichert haben. Das Modul ist fertig.

HINWEIS

Sie finden dieses Modul auch unter dem Namen *Toolsammlung* innerhalb der Begleitdateien zu diesem Buch, die Sie, wie in der Einleitung erläutert, kostenfrei aus dem Internet herunterladen können.

Sie können das Modul nun testweise importieren und prüfen, ob die Funktion *Get-SoftwareUpdates* anschließend zur Verfügung steht:

```
PS > import-module C:\kurs1\ToolSammlung
PS > Get-Command -module toolsammlung
```

CommandType	Name	Definition
-----	----	-----
Function	Get-SoftwareUpdates	...

```
PS > get-softwareupdates
```

Datum	Anwendung	Version	Herausgeber
-----	-----	-----	-----
17.02.2011 23:25:45	Windows Live Ess...	15.4.3502.0922	Microsoft Co...
15.02.2011 13:49:28	Windows Live Ess...	15.4.3502.0922	Microsoft Co...
15.02.2011 13:49:28	Mozilla Firefox ...	3.6.3 (de)	Mozilla
(...)			

Hintergrund

Selbsterstellte Module sind ein hervorragender Weg, um die Komplexität von PowerShell zu verbergen und neue einfache Cmdlet-ähnliche Befehle für andere bereitzustellen. Im Beispiel eben haben Sie gesehen, dass ein Modul im Grunde nur aus einem Ordner sowie einem »Inhaltsverzeichnis« besteht.

Als »Inhaltsverzeichnis« fungiert die *.psm1*-Datei im Modulordner, die unbedingt genauso heißen muss wie der Ordner selbst. Sie wird ausgeführt, wenn das Modul importiert wird, und kann alle übrigen Skripts nachladen.

Dabei steht *\$PSScriptRoot* für das Wurzelverzeichnis des Modulordners und sorgt dafür, dass die Dateireferenzen relativ zum aktuellen Ort des Modulordners angegeben werden können, das Modul also portabel bleibt.

Um erfolgreich eigene Module zu erstellen, gibt es eine Reihe von Best Practice-Regeln, die Sie kennen sollten:

- **Kein direkt ausführbarer Code in Skriptdateien** Skripts, die Sie in ein Modul einbinden, sollten ausschließlich Funktionsdefinitionen enthalten, aber keinen sofort ausführbaren Code. Ein Skript sollte also nicht selbst bereits eine Funktion aufrufen, denn genau das würde sonst später bei jedem Import des Moduls erneut geschehen. Weil während des Imports alle Bildschirmausgaben unterdrückt werden, würde dies zudem keinerlei Nutzen bringen und nur zu erheblichen Ladeverzögerungen führen.
- **Nur zugelassene Funktionsnamen** Sorgen Sie dafür, dass die Namen aller exportierten Funktionen der bei Cmdlets üblichen Verb-Substantiv-Syntax entsprechen, und wählen Sie als Verb ausschließlich zugelassene Verben, die Sie der Liste entnehmen können, die *Get-Verb* liefert. Andernfalls erscheint beim Import des Moduls jedes Mal eine Warnung.

- **Gewähren Sie Hilfe** Damit der Anwender Ihres Moduls zu Ihren Funktionen dieselbe komfortable Hilfe abrufen kann wie bei Cmdlets, sollte jede exportierte Funktion einen Block »Comment Based Help« enthalten. Wie dies geschieht, haben Sie bereits im Kapitel 6 erfahren.
- **Kein »Missbrauch« der .psm1-Datei** Da beim Import eines Moduls im Grunde nur die .psm1-Datei ausgeführt wird, könnte man die Dateierweiterung eines Skripts auch von .ps1 auf .psm1 ändern und in einem Ordner speichern. Sofern Skript und Ordner den gleichen Namen tragen, hätten Sie damit ein funktionstüchtiges Modul. Sinnvoll ist das aber nicht. Indem Sie die Skripts mit den eigentlichen Funktionen separat und eigenständig halten und nur über die .psm1-Datei nachladen, bleiben die Skripts einzeln wartbar und können separat getestet und debuggt werden.

Abschließend sollten Sie Ihr Modul mit einem Modul-Manifest ausstatten. Dieses Manifest legt allgemeine Informationen wie Version, Autor und Copyright fest und bestimmt außerdem, welche Funktionen, Aliase, Variablen und sonstigen Elemente Ihres Moduls öffentlich sein sollen. Ein Manifest wird mit *New-ModuleManifest* erstellt und in eine .psd1-Datei gespeichert, die im Modulordner liegt und genauso heißen muss wie dieser Ordner.

Sie können die notwendigen Angaben als Parameter an *New-ModuleManifest* übergeben oder interaktiv abfragen lassen. Nicht alle Parameter werden interaktiv abgefragt. Mit *-FunctionsToExport* legen Sie beispielsweise fest, welche Funktionen Ihrer Skripts für den Anwender später sichtbar sein sollen und können so Funktionen, die internen Zwecken dienen, verstecken.

```
PS > New-ModuleManifest -Path C:\kurs1\Toolsammlung\toolsammlung.psd1 ->
-ScriptsToProcess softwareupdates.ps1 -FunctionsToExport Get-Softwareupdates
```

```
Cmdlet New-ModuleManifest an der Befehlspipelineposition 1
Geben Sie Werte für die folgenden Parameter an:
NestedModules[0]:
Author: Tobias Weltner
CompanyName: scriptinternals
Copyright: 2011 Tobias Weltner
ModuleToProcess:
Description: Beispielm modul Buch Scripting für Administratoren
TypesToProcess[0]:
FormatsToProcess[0]:
RequiredAssemblies[0]:
FileList[0]:
```

TIPP

Ohne eine Manifestdatei können Sie von innerhalb der .psm1-Datei ebenfalls bestimmen, welche Funktionen, Aliase oder Variablen öffentlich sein sollen. Dazu setzen Sie *Export-ModuleMember* ein.

Zusammenfassung

Nicht jeder liebt Skriptprogrammierung. Deshalb ist PowerShell in unterschiedliche Schwierigkeitsgrade unterteilt, und wer sich allein auf den Einsatz von Cmdlets und der Pipeline beschränkt, kann mit sehr geringem Aufwand und ohne große Programmierkenntnisse höchst effektive Automationslösungen erstellen.

Module erweitern dieses Spektrum, indem sie neue Cmdlets und neue Provider nachrüsten, die in sich aber ganz genauso funktionieren wie die im Lieferumfang der PowerShell enthaltenen Basis-Cmdlets. Als Anwender braucht also nur die passende PowerShell-Erweiterung nachgeladen zu werden, um anschließend Active Directory, Exchange, VMware oder andere Bereiche zu verwalten.

Möglich ist dies, weil Cmdlets und Provider die Komplexität abschirmen und weil Module quasi neues »Fachwissen« liefern. Wie dies im Detail geschieht, haben Sie am Beispiel eines selbsterstellten Moduls erlebt. Mit einfachen Bordmitteln und nur wenigen Schritten kann ein Skript in ein Modul verwandelt werden. So lassen sich eigene Befehlserweiterungen verfassen. Module können aber auch dabei helfen, den eigenen Arbeitsplatz besser zu organisieren.

Anstatt Hunderte praktischer Skripts an verschiedensten Stellen zu lagern, um dann bei Bedarf das passende zu suchen, fasst man diese als Modul(e) zusammen. Es genügt dann, das für ein Thema passende Modul zu importieren, um sofort Zugriff auf alle praktischen Befehlserweiterungen zu haben, die man früher einmal entwickelt hatte.

Kapitel 18

PowerShell Remoting

In diesem Kapitel:

Cmdlets mit eigener Remoteunterstützung finden	508
PowerShell Remoting einrichten	510
Interaktiv auf ein Remotesystem zugreifen	512
Befehl oder Skript remote ausführen	513
Berechtigungen für PowerShell Remoting konfigurieren	518
Autostart-Skripts für Remoteverbindungen einrichten	519
PSSession öffnen und daraus Befehle importieren	521
Remotebefehle als Befehlserweiterung nutzen	523
Remoteauthentifizierung mit CredSSP einrichten	525
Zusammenfassung	528

Manche Cmdlets sind von sich aus remotefähig und unterstützen dann den Parameter *–Computername* sowie mitunter auch den Parameter *–Credential* zur Anmeldung unter einem anderen Benutzerkonto. Aber längst nicht alle Cmdlets bieten diese Remoteunterstützung.

Deshalb führt PowerShell 2.0 ein universelles Remotingkonzept ein, mit dessen Hilfe beliebige Befehle remotefähig werden. Voraussetzung dafür ist, dass sowohl Client als auch Server über PowerShell 2.0 verfügen und dass der Server seine Remotefunktionen aktiviert hat.

Jetzt kann der Client einzelne Befehle oder ganze Skripts automatisch auf den Server übertragen und dort in einer unsichtbaren PowerShell-Sitzung ausführen lassen. Die Ergebnisse werden ebenso automatisch zum Client zurücktransportiert. Der ausgeführte Code wird also hierbei eigentlich gar nicht remote ausgeführt, sondern lokal – nur eben auf einem anderen System. Aus diesem Grund lassen sich auf diese Weise beliebige Befehle auf anderen Systemen ausführen.

Cmdlets mit eigener Remoteunterstützung finden

Sie möchten wissen, welche Cmdlets über eingebaute Remoteunterstützung verfügen. Sie müssen beispielsweise auf ein Remotesystem zugreifen, das über keine PowerShell verfügt.

Lösung

Listen Sie mit *Get-Help* alle Cmdlets auf, die den Parameter *–Computername* unterstützen:

```
PS > Get-Help * -parameter ComputerName
```

Name	Category	Synopsis
----	-----	-----
Get-WinEvent	Cmdlet	Ruft Ereignisse aus Ereignisprot...
Get-Counter	Cmdlet	Ruft Leistungsindikatorendaten von...
Test-WSMan	Cmdlet	Testet, ob auf einem lokalen Com...
Invoke-WSManAction	Cmdlet	Ruft eine Aktion für das vom Res...
Connect-WSMan	Cmdlet	Stellt auf einem Remotecomputer ...
Disconnect-WSMan	Cmdlet	Trennt die Verbindung des Client...
Get-WSManInstance	Cmdlet	Zeigt Verwaltungsinformationen f...
Set-WSManInstance	Cmdlet	Ändert die Verwaltungsinformatio...
Remove-WSManInstance	Cmdlet	Löscht eine Verwaltungsressource...
New-WSManInstance	Cmdlet	Erstellt eine neue Instanz einer...
(...)		

Möchten Sie nur solche Cmdlets sehen, die gleichzeitig auch den Parameter *–Credential* unterstützen, mit dem man sich unter einem anderen Benutzernamen anmelden kann, lassen Sie sich von *Get-Command* alle Cmdlets ausgeben und filtern nach diesen beiden Parameternamen:

```
PS > Get-Command -commandType Cmdlet | Where-Object { →
    $_.Parameters.ContainsKey('credential') } | Where-Object { →
    $_.Parameters.ContainsKey('ComputerName') }
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Connect-WSMan	Connect-WSMan [[-ComputerNam...
Cmdlet	Enter-PSSession	Enter-PSSession [-ComputerNa...
Cmdlet	Get-HotFix	Get-HotFix [[-Id] <String[]>...
Cmdlet	Get-WinEvent	Get-WinEvent [[-LogName] <St...
Cmdlet	Get-WmiObject	Get-WmiObject [-Class] <Stri...
Cmdlet	Get-WSManInstance	Get-WSManInstance [-Resource...
Cmdlet	Invoke-Command	Invoke-Command [-ScriptBlock...
Cmdlet	Invoke-WmiMethod	Invoke-WmiMethod [-Class] <S...
Cmdlet	Invoke-WSManAction	Invoke-WSManAction [-Resourc...
Cmdlet	New-PSSession	New-PSSession [[-ComputerNam...
Cmdlet	New-WSManInstance	New-WSManInstance [-Resource...
Cmdlet	Register-WmiEvent	Register-WmiEvent [-Class] <...
Cmdlet	Remove-WmiObject	Remove-WmiObject [-Class] <S...
Cmdlet	Remove-WSManInstance	Remove-WSManInstance [-Resou...
Cmdlet	Restart-Computer	Restart-Computer [[-Computer...
Cmdlet	Set-WmiInstance	Set-WmiInstance [-Class] <St...
Cmdlet	Set-WSManInstance	Set-WSManInstance [-Resource...
Cmdlet	Stop-Computer	Stop-Computer [[-ComputerNam...
Cmdlet	Test-Connection	Test-Connection [-ComputerNa...
Cmdlet	Test-WSMan	Test-WSMan [[-ComputerName] ...

Hintergrund

Sie erhalten auf diese Weise lediglich eine Liste aller Cmdlets, die die angegebenen Parameter unterstützen. Sie alle sind remotefähig, aber nur ein Teil davon verwendet eine eigene Remoting-Funktionalität. Der andere Teil entspricht den Cmdlets, die das universelle PowerShell Remoting zur Verfügung stellen.

Um nur die Cmdlets zu identifizieren, die über ein eigenes Remoting verfügen, könnten Sie aber in der Beschreibung des Parameters *-Computername* nach dem Stichwort »beruht nicht auf« suchen (passen Sie das Stichwort auf nicht-deutschen Systemen entsprechend an):

```
PS > Get-Help * -parameter Computername | Where-Object { $_ | Get-Help →
    -parameter Computername | Out-String | Select-String 'beruht nicht auf' }
```

Name	Category	Synopsis
----	-----	-----
Get-WinEvent	Cmdlet	Ruft Ereignisse aus Ereignisprot...
Get-Counter	Cmdlet	Ruft Leistungsindikatordaten von...
Get-EventLog	Cmdlet	Ruft die Ereignisse in einem Ere...
Clear-EventLog	Cmdlet	Löscht alle Einträge aus angegeb...
Write-EventLog	Cmdlet	Schreibt ein Ereignis in ein Ere...
Limit-EventLog	Cmdlet	Legt die Ereignisprotokolleigens...
Show-EventLog	Cmdlet	Zeigt die Ereignisprotokolle des...
New-EventLog	Cmdlet	Erstellt ein neues Ereignisproto...

Remove-EventLog	Cmdlet	Löscht ein Ereignisprotokoll ode...
Get-WmiObject	Cmdlet	Ruft Instanzen von WMI-Klassen (...)
Get-Process	Cmdlet	Ruft die Prozesse ab, die auf de...
Remove-WmiObject	Cmdlet	Löscht eine Instanz einer vorhan...
Get-Service	Cmdlet	Ruft die Dienste auf einem lokal...
Set-Service	Cmdlet	Startet, beendet und hält einen ...
Set-WmiInstance	Cmdlet	Erstellt oder aktualisiert eine ...
Get-HotFix	Cmdlet	Ruft die Hotfixes ab, die auf de...
Test-Connection	Cmdlet	Sendet ICMP-Echoanforderungspake...
Restart-Computer	Cmdlet	Startet das Betriebssystem auf d...
Stop-Computer	Cmdlet	Beendet lokale und Remotecompute...

Entsprechend liefert die Negation der Bedingung alle Cmdlets, die für das PowerShell Remoting eingesetzt werden:

```
PS > Get-Help * -parameter Computername | Where-Object { !($_ | Get-Help -parameter Computername | Out-String | Select-String 'beruht nicht auf') }
```

Name	Category	Synopsis
----	-----	-----
Test-WSMan	Cmdlet	Testet, ob auf einem lokalen Com...
Invoke-WSManAction	Cmdlet	Ruft eine Aktion für das vom Res...
Connect-WSMan	Cmdlet	Stellt auf einem Remotecomputer ...
Disconnect-WSMan	Cmdlet	Trennt die Verbindung des Client...
Get-WSManInstance	Cmdlet	Zeigt Verwaltungsinformationen f...
Set-WSManInstance	Cmdlet	Ändert die Verwaltungsinformatio...
Remove-WSManInstance	Cmdlet	Löscht eine Verwaltungsressource...
New-WSManInstance	Cmdlet	Erstellt eine neue Instanz einer...
Invoke-Command	Cmdlet	Führt Befehle auf lokalen Comput...
New-PSSession	Cmdlet	Erstellt eine dauerhafte Verbind...
Get-PSSession	Cmdlet	Ruft die Windows PowerShell-Sitz...
Remove-PSSession	Cmdlet	Schließt eine oder mehrere Windo...
Receive-Job	Cmdlet	Ruft die Ergebnisse der Windows ...
Enter-PSSession	Cmdlet	Startet eine interaktive Sitzung...
Invoke-WmiMethod	Cmdlet	Ruft Methoden der Windows-Verwal...
Register-WmiEvent	Cmdlet	Abonniert ein WMI-Ereignis (Wind...

PowerShell Remoting einrichten

Sie möchten einen Computer so konfigurieren, dass man auf ihn über das PowerShell Remoting remote zugreifen kann.

Lösung

Aktivieren Sie die Remoting-Funktionen mit *Enable-PSRemoting*:

```
PS > Enable-PSRemoting
```

Hierfür sind Administratorrechte erforderlich. Das Cmdlet meldet, welche Konfigurationsschritte es durchführen wird, und wartet auf eine Bestätigung durch Sie. Anschließend kann der Computer remote angesprochen werden.

Hintergrund

Enable-PSRemoting konfiguriert den Computer dauerhaft. Dieser Befehlsaufruf muss also nicht in jeder Session wiederholt werden. Die Remoting-Fähigkeiten bleiben so lange erhalten, bis Sie sie mit *Disable-PSRemoting* wieder abschalten.

Die Einrichtung erfordert Administratorrechte und muss deshalb bei aktivierter Benutzerkontensteuerung in einer PowerShell ausgeführt werden, die Sie mit *Als Administrator starten* geöffnet haben. Verfügt Ihr Benutzerkonto über kein Kennwort, kann die Konfiguration nicht durchgeführt werden.

TIPP

Möchten Sie die Konfigurationsänderungen unbeaufsichtigt ohne Sicherheitsabfrage durchführen, geben Sie den Parameter *-Force* an. Sollte es während der Konfiguration zu Firewallfehlern kommen, ist mindestens eine Ihrer Netzwerkverbindungen auf »öffentlich« eingestellt. In diesem Fall kann keine Firewallausnahme eingerichtet werden, die jedoch für PowerShell Remoting erforderlich ist. Ändern Sie entweder den Status der öffentlichen Netzwerkverbindung im Netzwerk- und Freigabecenter der Systemsteuerung oder deaktivieren Sie die Netzwerkverbindung vorübergehend und führen Sie *Enable-PSRemoting* erneut aus. Insbesondere virtuelle Netzwerkadapter werden häufig als »öffentlich« konfiguriert und lassen sich nicht umkonfigurieren. Solche Netzwerkverbindungen müssen daher während der Ausführung von *Enable-PSRemoting* in jedem Fall vorübergehend deaktiviert werden.

Dabei werden diese Änderungen vorgenommen:

- **WinRM-Dienst** Der WinRM-Dienst wird gestartet und sein Startmodus auf automatischen Start eingestellt. Dieser Dienst empfängt später die Remoteanforderungen. Auf Servern läuft dieser Dienst normalerweise ohnehin, auf Clients nicht.
- **Listener** Im WinRM-Dienst wird ein »Listener« registriert, der am Port 5985 auf Anfragen anderer Computer reagiert und diese an PowerShell weiterleitet
- **Firewall-Ausnahme** In der lokalen Windows-Firewall wird eine entsprechende Ausnahmeregel aktiviert, damit einlaufende Anfragen an Port 5985 empfangen werden können

Schalten Sie das Remoting mit *Disable-PSRemoting* wieder aus, wird nur der Listener entfernt. Die Konfiguration des WinRM-Diensts wird nicht geändert und auch die Firewallausnahme bleibt aktiv.

PowerShell Remoting muss in einer Domänenumgebung nur auf dem Server aktiviert werden, also bei dem Computer, auf den zugegriffen werden soll. In Peer-to-Peer-Umgebungen oder Multidomänen muss PowerShell Remoting auch auf dem Client vorübergehend aktiviert werden, um diesen korrekt zu konfigurieren.

Die Konfiguration des Clients ist immer dann erforderlich, wenn die Anmeldung am Zielcomputer nicht über Kerberos erfolgt – entweder also, weil kein Active Directory verfügbar ist, oder weil Sie nicht den Rechnernamen des Zielcomputers verwenden, sondern seine IP-Adresse. Führen Sie in diesem Fall auf dem Client diese Konfigurationsschritte durch:

```
PS > Enable-PSRemoting -force
PS > Set-Item wsman:\localhost\client\trustedhosts * -force
PS > Disable-PSRemoting -force
```

HINWEIS

Die Remotingkonfiguration kann alternativ auch über Gruppenrichtlinien zentral für das gesamte Unternehmen eingerichtet werden.

Interaktiv auf ein Remotesystem zugreifen

Sie möchten Ihre PowerShell-Konsole vorübergehend mit einem Remotesystem verbinden, zum Beispiel, um dort Änderungen vorzunehmen.

Lösung

Setzen Sie *Enter-PSSession* ein, und geben Sie den Namen des Zielsystems an. Auf dem Zielsystem muss PowerShell vorhanden und PowerShell Remoting aktiviert worden sein:

```
PS > Enter-PSSession storage1
[storage1]: PS C:\Users\Administrator\Documents>
```

Nach erfolgreicher Verbindungsaufnahme ändert sich die Eingabeaufforderung und zeigt in eckigen Klammern den Namen des verbundenen Systems an. Auch der übrige Prompt kann sich ändern, weil Sie sich nun in einer anderen PowerShell-Session befinden. Ist er Ihnen zu lang, kürzen Sie ihn durch Überschreiben der Funktion *prompt*:

```
[storage1]: PS C:\Users\Administrator\Documents> function prompt { "PS > " }
```

Alle Befehle, die Sie nun eingeben, werden auf dem Remotesystem ausgeführt. Im folgenden Beispiel wird eine Textdatei auf dem Remotesystem angelegt:

```
[storage1]: PS > $env:computername
STORAGE1
[storage1]: PS > "Test" | Out-File c:\testdatei.txt
```

Um die Remotesitzung zu beenden und zu Ihrer lokalen Sitzung zurückzukehren, geben Sie *Exit-PSSession* ein:

```
[storage1]: PS > Exit-PSSession
PS >
```


Hintergrund

Für die Verbindung zu einem Remotesystem sind Administratorrechte erforderlich. Sie brauchen sich am Remotesystem nicht anzumelden, wenn ...

- ... Ihr Computer und der Zielcomputer Mitglied derselben Domäne sind und Sie aktuell mit einem Konto angemeldet sind, das auf dem Zielsystem über Administratorrechte verfügt, und wenn Sie den Zielcomputernamen und nicht seine IP-Adresse angegeben haben
- ... Ihr Computer und der Zielcomputer beide nicht in einer Domäne angemeldet sind und es auf dem Zielcomputer ein Benutzerkonto mit Administratorrechten gibt, dessen Name genauso lautet wie Ihr eigener Benutzername

In allen anderen Fällen ist eine explizite Anmeldung notwendig:

```
PS > Enter-PSSession 192.168.2.234
```

```
Enter-PSSession : Beim Verbinden mit dem Remoteserver ist folgender Fehler aufgetreten:  
Zugriff verweigert Weitere Informationen finden Sie im Hilfethema  
"about_Remote_Troubleshooting".
```

```
PS > Enter-PSSession 192.168.2.234 -Credential powershell\Tobias  
[192.168.2.234]: PS C:\Users\Tobias\Documents> Exit-PSSession  
PS >
```

ACHTUNG *Enter-PSSession* kann nicht dazu verwendet werden, um ein Skript remote auszuführen, denn *Enter-PSSession* überträgt nur die interaktiv eingegebenen Befehle an das Remotesystem. Wenn Sie also in einem Skript *Enter-PSSession* verwenden, wird das Skript nach wie vor lokal ausgeführt. Verwenden Sie stattdessen *Invoke-Command* mit dem Parameter *-File*. Starten Sie außerdem keine Programme, die ein Fenster öffnen. Weil Ihre Befehle auf dem Zielsystem in einer unsichtbaren Konsole ausgeführt werden, wäre das Fenster ebenfalls unsichtbar und könnte Ihre Sitzung sogar blockieren. *Enter-PSSession* ist nur zur interaktiven Ausführung konsolenbasierter Befehle gedacht.

Befehl oder Skript remote ausführen

Sie möchten einen Befehl oder ein ganzes Skript auf einem anderen System ausführen.

Lösung

Verwenden Sie *Invoke-Command* und übergeben Sie dem Cmdlet entweder einen Skriptblock mit dem auszuführenden Befehl oder verwenden Sie den Parameter *-File*, um ein PowerShell-Skript anzugeben, das sich auf Ihrem Computer befindet.

Die folgende Zeile legt auf dem Computer *storage1* einen neuen Registrierungsschlüssel in *HKEY_LOCAL_MACHINE\Software* namens *NewKey* an und weist ihm den Standardwert *New-Value* zu:

```
PS > Invoke-Command { New-Item HKLM:\Software\NewKey -value 'New value' } ->
-computername storage1
```

Hive: HKEY_LOCAL_MACHINE\Software

SKC	VC Name	Property	PSComputerName
---	----	-----	-----
0	1 NewKey	{(default)}	storage1

Möchten Sie das lokal gespeicherte Skript `c:\kurs1\job12.ps1` auf einem Remotesystem ausführen, gehen Sie so vor:

```
PS > Invoke-Command -file c:\kurs1\job12.ps1 -computername storage1
```

TIPP

Melden Sie sich am Zielsystem mit `-Credential` explizit an, falls Sie Fehlermeldungen über mangelnde Zugriffsrechte erhalten. Falls Ihre Computer nicht Mitglied derselben Domäne sind oder Sie gar keine Domäne einsetzen, muss am Client einmalig mit Administratorrechten die folgende Einstellung gesetzt werden:

```
PS > Set-Item wsman:\localhost\client\trustedhosts * -force
```

Eine Anmeldung ist nicht erforderlich, wenn beide Computer Mitglied derselben Domäne sind, Ihr Konto über Administratorrechte auf dem Zielsystem verfügt und Sie den Rechnernamen und nicht seine IP-Adresse angegeben haben oder falls beide Systeme kein Mitglied irgendeiner Domäne sind und es auf dem Zielsystem ein Benutzerkonto mit lokalen Administratorrechten gibt, das so heißt wie Ihr eigener Benutzername.

Hintergrund

`Invoke-Command` kann Code zu einem oder mehreren Remotesystemen senden. Dazu übergeben Sie den Code entweder direkt als Skriptblock in geschweiften Klammern oder Sie geben den Pfad zu einer lokalen Skriptdatei an. In diesem Fall liest PowerShell den darin enthaltenen Code und sendet ihn als Skriptblock an das Remotesystem.

Die Ergebnisse des Codes werden automatisch zu Ihrer lokalen Sitzung zurücktransportiert. Wenn Sie also Code remote ausführen und an den Ergebnissen interessiert sind, dürfen Sie diese nicht etwa auf dem Remotesystem speichern:

```
PS > Invoke-Command { Get-EventLog System -EntryType Error -Newest 10 |
Export-CSV $env:temp\liste.csv } -computer storage1
```

Verschieben Sie stattdessen den Exportbefehl etwas weiter nach rechts, sodass die Daten erst dann exportiert werden, wenn sie zurück zu Ihrem lokalen System transportiert wurden. Die CSV-Datei liegt jetzt nicht mehr auf dem Remotesystem, sondern auf Ihrem eigenen:

```
PS > Invoke-Command { Get-EventLog System -EntryType Error -Newest 10 } →
-computer storage1 | Export-CSV $env:temp\liste.csv
```

Beim Rücktransport vom Remotesystem zu Ihrem System werden die Ergebnisobjekte vorübergehend in XML verwandelt und nach Eintreffen auf Ihrem System wieder in Objekte zurückverwandelt. Hierbei verlieren die Objekte sämtliche Methoden (Befehle) und jede Verbindung zum Remotesystem. Sie erhalten also nur noch »Read-only«-Kopien. Diese Beschränkung existiert bei einer interaktiven Remotesitzung (*Enter-PSSession*) nicht und kann auch bei *Invoke-Command* leicht umgangen werden, indem man nötigenfalls die Methoden von Ergebnisobjekten noch innerhalb des Remotecodes auswertet.

Möchten Sie Code auf mehreren Computern gleichzeitig ausführen, geben Sie die Computernamen als kommaseparierte Liste an. Die folgenden Zeilen würden den Remotedesktop auf drei Computern freischalten:

```
$code = {
>> $key = "HKLM:\SYSTEM\CurrentControlSet\Control\Terminal Server"
>> Set-ItemProperty $key fDenyTSConnections 0
>> }
>>
Invoke-Command $code -computer 192.168.2.234, 192.168.2.115, storage1 →
-credential powershell\Tobias
```

Denken Sie immer daran, dass der Code, den Sie angeben, zunächst auf das Remotesystem übertragen und erst dann ausgeführt wird. Auf dem Remotesystem müssen also alle Voraussetzungen erfüllt sein, um den Code auch tatsächlich ausführen zu können. Ist der Code zum Beispiel von geladenen PowerShell-Erweiterungen oder anderen Funktionen abhängig, sind diese möglicherweise auf dem Zielsystem (noch) nicht geladen oder vorhanden und der Code schlägt fehl.

Dasselbe gilt für Variablen, die in Ihrer lokalen Sitzung vielleicht definiert sind, auf dem Zielsystem aber nicht. Die folgende Funktion funktioniert lokal, aber nicht remote:

```
Function Get-ErrorEvents($logfile) {
    Get-EventLog $logfile -EntryType Error -Newest 10
}
```

```
PS > Get-ErrorEvents System
```

Index	Time	EntryType	Source	InstanceID	Message
----	----	-----	-----	-----	-----
180316	Feb 18 12:39	Error	volsnap	3221618723	Die Sch...
180267	Feb 18 10:02	Error	Service Contro...	3221232483	Das Zei...
180266	Feb 18 10:02	Error	Service Contro...	3221232483	Das Zei...

```
PS > Invoke-Command { Get-ErrorEvents System } -computername storage1
```

Die Benennung "Get-ErrorEvents" wurde nicht als Name eines Cmdlet, einer Funktion, einer Skriptdatei oder eines ausführbaren Programms erkannt. Überprüfen Sie die Schreibweise des Namens, oder ob der Pfad korrekt ist (sofern enthalten), und wiederholen Sie den Vorgang.

Die Fehlermeldung ist verständlich, denn Sie haben die Funktion nicht mit an das Remotesystem übertragen. Der korrekte Aufruf wäre:

```
$code = {
Function Get-ErrorEvents($logfile) {
    Get-EventLog $logfile -EntryType Error -Newest 10
}

Get-ErrorEvents System
}

Invoke-Command $code -computername storage1
```

Mit *Invoke-Command* können Sie Funktionen, die normalerweise nur lokal ausführbar sind, remotefähig machen. Allerdings sind auch hierbei einige Details zu beachten. Damit die Funktion *Get-ErrorEvents* remotefähig wird, legt man einen Parameter *-Computername* an. Gibt der Anwender einen Computernamen an, lässt man die Ereignisdaten vom Remotesystem abrufen, sonst nicht:

```
Function Get-ErrorEvents($logfile='System', $computername=$null) {
    If ($computername -ne $null) {
        Invoke-Command { Get-EventLog $logfile -EntryType Error -Newest 10 } →
        -computername $computername
    } else {
        Get-EventLog $logfile -EntryType Error -Newest 10
    }
}
```

Die Funktion arbeitet lokal einwandfrei, aber wenn ein Computername angegeben wird, kommt es zu einem Fehler:

PS > **Get-ErrorEvents**

Index	Time	EntryType	Source	InstanceID	Message
180316	Feb 18 12:39	Error	volsnap	3221618723	Die Sch...
180267	Feb 18 10:02	Error	Service Control...	3221232483	Das Zei...

(...)

PS > **Get-ErrorEvents -comp storage1**

Das Argument kann nicht an den Parameter "LogName" gebunden werden, da es NULL ist.

Der Grund: Der Skriptblock, der an das Remotesystem übermittelt wird, enthält die Variable *\$logname*, die angeben soll, welches Logbuch auszulesen ist. Diese Variable existiert aber nur auf dem lokalen System, nicht auf dem Remotesystem.

Um lokale Variablen an einen Skriptblock zu übergeben, definieren Sie im Skriptblock Parameter und übergeben dann die Parameter mit *-ArgumentList*:

```
Function Get-ErrorEvents($logfile='System', $computername=$null) {
    If ($computername -ne $null) {
        Invoke-Command {
            param($logfile)
            Get-EventLog $logfile -EntryType Error -Newest 10
        } -computername $computername -ArgumentList $logfile
    } else {
        Get-EventLog $logfile -EntryType Error -Newest 10
    }
}
```

Nun funktioniert die Funktion einwandfrei, sowohl lokal als auch remote:

PS > **Get-ErrorEvents**

Index	Time	EntryType	Source	InstanceID	Message
180316	Feb 18 12:39	Error	volsnap	3221618723	Die Sch...
180267	Feb 18 10:02	Error	Service Control...	3221232483	Das Zei...

(...)

PS > **Get-ErrorEvents -comp storage1**

Index	Time	EntryType	Source	InstanceID	Message	PSComputerName
4426	Feb 14 22:30	Error	EventLog	2147489656	Da...	s...
4361	Feb 07 09:41	Error	iaStor	3221487625	Da...	s...

(...)

Allerdings unterscheiden sich die Ausgaben noch. Werden Ergebnisse aus einer Remotesitzung zurückgeliefert, fügt PowerShell automatisch einige neue Eigenschaften an, aus denen hervorgeht, von welchem Computer und aus welcher Session die Daten stammen. In der Praxis sollte man deshalb die erwünschten Eigenschaften mit *Select-Object* festlegen:

```
Function Get-ErrorEvents($logfile='System', $computername=$null) {
    & {
        If ($computername -ne $null) {
            Invoke-Command {
                param($logfile)
                Get-EventLog $logfile -EntryType Error -Newest 10
            } -computername $computername -ArgumentList $logfile
        } else {
            Get-EventLog $logfile -EntryType Error -Newest 10
        }
    } | Select-Object TimeWritten, EntryType, Source, Message
}
```

PS > **Get-ErrorEvents**

TimeWritten	EntryType	Source	Message
18.02.2011 12:39:40	Error	volsnap	Die Schattenkopi...
18.02.2011 10:02:53	Error	Service Control ...	Das Zeitlimit (3...
18.02.2011 10:02:23	Error	Service Control ...	Das Zeitlimit (3...
(...)			

PS > **Get-ErrorEvents -comp storage1**

TimeWritten	EntryType	Source	Message
14.02.2011 22:30:27	Error	EventLog	Das System wurde...
07.02.2011 09:41:46	Error	iaStor	Das Gerät \Devic...
29.01.2011 03:04:47	Error	DCOM	Bei DCOM ist der...
(...)			

Berechtigungen für PowerShell Remoting konfigurieren

Sie möchten festlegen, welche Personen sich via PowerShell Remoting mit einem Zielsystem verbinden dürfen.

Lösung

Die Berechtigungen einer Remoteverbindung werden über sogenannte »SessionConfigurations« festgelegt. Um diese verwalten zu können, benötigen Sie Administratorrechte. Die Konfigurationen befinden sich jeweils auf dem Zielsystem, auf das zugegriffen werden soll, und müssen dort konfiguriert werden.

Die Standardkonfiguration, die als Vorgabe verwendet wird, heißt *Microsoft.PowerShell*. Die Berechtigungseinstellungen kann man mit *Set-PSSessionConfiguration* in einem Dialogfeld anzeigen lassen und dort auch ändern:

PS > **Set-PSSessionConfiguration Microsoft.PowerShell -ShowSecurityDescriptorUI**

Hintergrund

Alle Einstellungen rund um Remoteverbindungen werden in Datensätzen zusammengefasst, die »SessionConfiguration« heißen. Wird bei der Verbindungsaufnahme keine besondere Konfiguration mit dem Parameter *-ConfigurationName* angegeben, verwendet PowerShell die Einstellungen aus der Konfiguration *Microsoft.PowerShell*, beziehungsweise diejenigen, die in der Variablen *\$PSSessionConfigurationName* vorgegeben sind:

PS > **\$PSSessionConfigurationName**
<http://schemas.microsoft.com/powershell/Microsoft.PowerShell>

Die Konfiguration definiert auf dem Zielsystem, wie die Remoteverbindung hergestellt werden soll, und legt zum Beispiel fest, wer die Konfiguration überhaupt verwenden darf. Als Vorgabe dürfen nur lokale Administratoren Remoteverbindungen herstellen, aber indem Sie die Sicherheitsbeschreibung ändern, können Sie Remoteverbindungen auch für normale Anwender gestatten oder auf nur einen ausgewählten Personenkreis weiter einschränken.

Autostart-Skripts für Remoteverbindungen einrichten

Sie möchten, dass bei der Herstellung einer Remotesession zu einem Computer automatisch ein Autostart-Skript ausgeführt wird, zum Beispiel, damit automatisch Module nachgeladen oder die Eingabeaufforderung angepasst oder verkürzt werden kann.

Lösung

Ändern Sie die *PSSessionConfiguration* und legen Sie darin fest, welches Skript bei der Verbindungsaufnahme gestartet werden soll.

Richten Sie sich mit *Register-PSSessionConfiguration* eine neue *PSSessionConfiguration* ein, wenn Sie die Standardkonfiguration nicht verändern wollen:

```
PS > Register-PSSessionConfiguration -Name Interaktiv -StartupScript →
$home\remoteprofile.ps1
```

Bestätigung

Möchten Sie diese Aktion wirklich ausführen?

Ausführen des Vorgangs "Register-PSSessionConfiguration" für das Ziel "Name: Interaktiv. Erlaubt Administratoren die Remoteausführung von Windows PowerShell-Befehlen auf diesem Computer."

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe
(Standard ist "J"):j

```
WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin
```

Name	Type	Keys
----	----	----
Interaktiv	Container	{Name=Interaktiv}

Bestätigung

Möchten Sie diese Aktion wirklich ausführen?

Ausführen des Vorgangs ""Restart-Service"" für das Ziel "Name: WinRM".

[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe
(Standard ist "J"):j

Dieser Befehl legt eine neue Konfiguration namens *Interaktiv* an, die ein Skript im Benutzerprofil des Anwenders namens *remoteprofile.ps1* ausführt. Damit ein Autoskript tatsächlich gestartet werden kann, muss die ExecutionPolicy des Systems dies auch zulassen. Überprüfen Sie deshalb außerdem die Richtlinie und erlauben Sie falls nötig die Skriptausführung:

```
PS > Set-ExecutionPolicy RemoteSigned -force
```

Legen Sie das Autostart-Skript an und vermerken Sie darin, was beim Anmeldevorgang geschehen soll. Wünschen Sie sich zum Beispiel eine kürzere Eingabeaufforderung in interaktiven Sitzungen, könnten Sie darin die Funktion *prompt* umdefinieren:

```
Function prompt { 'PS > ' }
```

Möchte sich der Anwender mit der neuen Konfiguration *Interaktiv* verbinden, ruft er von einem anderen Computer aus die neue Konfiguration auf dem Zielsystem ab:

```
PS > enter-psession storage1 -ConfigurationName Interaktiv
[storage1]: PS >
[storage1]: PS > exit-psession
```

Der Unterschied ist augenfällig. Meldet sich der Anwender mit der Standardkonfiguration an, erhält er wieder die ausladende Standardeingabeaufforderung:

```
PS > enter-psession storage1
[storage1]: PS C:\Users\Administrator\Documents>
[storage1]: PS C:\Users\Administrator\Documents> Exit-PSSession
```

Hintergrund

Mit *Register-PSSessionConfiguration* wird eine neue Konfiguration angelegt und konfiguriert. Möchten Sie eine vorhandene Konfiguration ändern oder erweitern, wählen Sie *Set-PSSessionConfiguration* und geben den Namen der vorhandenen Konfiguration an.

Alle vorhandenen Konfigurationen listet *Get-PSSessionConfiguration* auf:

```
PS > Get-PSSessionConfiguration
```

Name	PSVersion	StartupScript	Permission
----	-----	-----	-----
Interaktiv	2.0	C:\Users\w7-pc9\r...	
Microsoft.PowerShell	2.0		
Microsoft.PowerShell32	2.0		

Die Einstellungen einer Konfiguration machen Sie sichtbar, indem Sie sie an *Select-Object* * weiterleiten. Die nächste Zeile macht die Einstellungen der neuen Konfiguration sichtbar und blendet dazu alle störenden Standardeigenschaften aus:


```
PS > Get-PSSessionConfiguration Interaktiv | Select-Object * -exclude xmlns, filename, sdkversion, xmlrenderingtype, lang, resourceuri, supportoptions, exactmatch
```

```
Name           : Interaktiv
PSVersion      : 2.0
startupscript  : C:\Users\w7-pc9\remoteprofile.ps1
Capability     : {Shell}
Permission     :
```

Die Verwaltung von Konfigurationen muss immer auf dem System erfolgen, wo sie gelten sollen, also auf dem Zielsystem, das später über Remoting angesprochen werden soll. Über PowerShell können Konfigurationen remote normalerweise nicht angelegt werden, weil dazu die Anmelde-Informationen aus der Remotesitzung heraus weitergegeben werden müssten. Das ist nur möglich, wenn die Weiterleitung mit CredSSP gestattet wurde.

Zumindest kann man die Konfigurationen aber remote anzeigen lassen, um herauszufinden, welche zur Verfügung stehen. Die folgenden Zeilen verbinden sich zum System *storage1* und listen die dort definierten Konfigurationen auf:

```
PS > Connect-WSMan storage1
PS > Dir wsman:\storage1\plugin\* | Where-Object { (Get-Item ('{0}\Filename' -f $_.PSPath)).Value -like '*\pwrshplugin.dll' }
```

```
WSManConfig: Microsoft.WSMan.Management\WSMan::storage1\Plugin
```

Name	Type	Keys
----	----	----
Interaktiv	Container	{Name=Interaktiv}
Microsoft.PowerShell	Container	{Name=Microsoft.PowerShell}

PSSession öffnen und daraus Befehle importieren

Sie möchten eine PowerShell Remoting-Session so lange geöffnet halten, wie Sie wollen. Sie möchten sich zum Beispiel interaktiv mit einem Remotesystem verbinden, dann dort einen länger dauernden Vorgang starten, sich wieder abmelden und erst später nachschauen, ob der Vorgang abgeschlossen ist. Oder Sie möchten aus einer Session Remotebefehle in Ihre lokale Session einblenden.

Lösung

Legen Sie eine dauerhafte PSSession mit *New-PSSession* an. Diese Session bleibt so lange geöffnet, bis Sie sie mit *Remove-PSSession* wieder freigeben:

```
PS > $session = New-PSSession -computer 192.168.2.234 -credential powershell\Administrator
```

Sie können diese Session nun beliebig oft verwenden, solange sie geöffnet ist. Zum Beispiel können Sie sich interaktiv mit ihr verbinden und dann auf dem Remotesystem ein Modul nachladen.

```
PS > Enter-PSSession -Session $session
[192.168.2.234]: PS C:\Users\Administrator\Documents> import-module servermanager
[192.168.2.234]: PS C:\Users\Administrator\Documents> exit-pssession
PS >
```

Auch nachdem Sie in Ihre lokale Session zurückgekehrt sind, bleibt die Remotesession aktiv und enthält nun die Cmdlets aus dem nachgeladenen Modul, beispielsweise dem Server-Manager von Windows Server 2008 R2.

Sie könnten nun die Cmdlets dieses Moduls aus der Remotesession mit *Import-PSSession* in Ihre eigene lokale Session einblenden. Dies nennt man auch implizites Remoting:

```
PS > $import = Import-PSSession $session -Module Servermanager
PS > $import.ExportedCommands
```

Name	Value
Remove-WindowsFeature	Remove-WindowsFeature
Get-WindowsFeature	Get-WindowsFeature
Add-WindowsFeature	Add-WindowsFeature

Sie könnten den Remoteserver nun direkt von Ihrem lokalen System aus verwalten. Obwohl das Modul *Servermanager* also auf Ihrem eigenen Computer überhaupt nicht vorhanden ist, stehen die Cmdlets daraus dennoch in Ihrer lokalen Konsole zur Verfügung.

Um beispielsweise zu prüfen, ob der PowerShell-Editor *ISE* auf dem Server installiert ist, und ihn gegebenenfalls nachzuinstallieren, verwenden Sie diesen Code:

```
PS > Get-WindowsFeature powershell-ise | Select-Object -expand Installed
False
PS > $ise = Get-WindowsFeature powershell-ise | Select-Object -expand Installed
PS > if (!$ise) { Add-WindowsFeature powershell-ise } else →
    { 'ISE ist installiert' }

RunspaceId      : 4a2ed449-f4d3-4cbd-a344-07cf17e4d0a5
Success         : True
RestartNeeded   : No
FeatureResult    : {Windows PowerShell Integrated Scripting Environment (ISE)}
ExitCode        : Success

ISE ist installiert
```

Hintergrund

Geben Sie bei *Enter-PSSession* oder *Invoke-Command* einen Computernamen an, legt PowerShell automatisch eine PSSession an und entfernt sie wieder, sobald das Cmdlet seine Arbeit ver-

richtet hat. Für viele Aufgaben ist das ein effizienter Weg. Insbesondere besteht dabei nicht die Gefahr, dass PSSessions »vergessen« werden und Serverressourcen in Beschlag nehmen, denn die maximale Anzahl gleichzeitig geöffneter Sessions ist begrenzt.

In manchen Fällen ist es allerdings wünschenswert, nicht jedes Mal eine neue PSSession anzulegen, zum Beispiel, wenn Sie den Zustand einer Session bewahren müssen oder wie im Beispiel gezeigt Cmdlets aus einer Remotesession in eine lokale Session einblenden wollen. In diesem Fall übernehmen Sie mit *New-PSSession* die Kontrolle über die Lebensspanne der Session und sind auch dafür verantwortlich, die Session nach Gebrauch mit *Remove-PSSession* wieder freizugeben.

Remotebefehle als Befehlserweiterung nutzen

Sie möchten Cmdlets, die eigentlich auf einem Remotesystem vorhanden sind, als Modul nachladen und sofort nutzen können.

Lösung

Überprüfen Sie, ob es auf dem Remotesystem eine Konfiguration gibt, in der die gewünschten Befehle vorhanden sind. Handelt es sich beispielsweise um einen Exchange-Server, lautet der Name dieser Konfiguration *Microsoft.Exchange*.

Sie können aber auch auf dem Remotesystem eine eigene Konfiguration definieren und zum Beispiel bei Windows Server 2008 R2 in dieser Konfiguration das Modul *Serververwaltung* laden (siehe den Abschnitt »Remoteauthentifizierung mit CredSSP einrichten« ab Seite 525).

Öffnen Sie dann eine neue PSSession mit der passenden Konfiguration zum Zielsystem und exportieren Sie die gewünschten Befehle aus der PSSession in ein Modul mit *Export-PSSession*.

Im folgenden Beispiel wurde die Konfiguration *ServerVerwaltung* aus dem Abschnitt »Remoteauthentifizierung mit CredSSP einrichten« verwendet. Mit den folgenden Zeilen erstellen Sie dann ein Modul, das die Cmdlets aus dem Modul *ServerManager* auf dem Remotesystem jederzeit verfügbar macht:

```
PS > $session = New-PSSession -computername 192.168.2.234 ->
-Cred powershell\Tobias -ConfigurationName ServerVerwaltung
PS > Export-PSSession $session -Module ServerManager ->
-OutputModule Server12Verwaltung
```

Verzeichnis: C:\Users\w7-pc9\Documents\WindowsPowerShell\Modules\Server12Verwaltung

Mode	LastWriteTime	Length	Name
-a---	18.02.2011 17:05	15608	Server12Verwaltung.psm1
-a---	18.02.2011 17:05	99	Server12Verwaltung.format.ps1xml
-a---	18.02.2011 17:05	603	Server12Verwaltung.psd1

Um die Cmdlets aus dem Modul *Server12Verwaltung* auf dem Remotesystem auszuführen, müssen Sie nun lediglich aus einer lokalen PowerShell-Sitzung heraus das soeben erstellte Modul importieren:

```
PS > Import-Module Server12Verwaltung
```

Die Cmdlets des Moduls *ServerManager* stehen sofort zur Verfügung:

```
PS > Get-Command -Module Server12Ver*
```

CommandType	Name	Definition
-----	----	-----
Function	Add-WindowsFeature	...
Function	Get-WindowsFeature	...
Function	Remove-WindowsFeature	...

Sobald Sie einen davon einsetzen, wird automatisch die PSSession zum Remotesystem aufgebaut und der Befehl remote ausgeführt:

```
PS > Get-WindowsFeature PowerShell-ISE
```

Neue Sitzung für implizite Remotevorgänge des Befehls "Get-WindowsFeature" wird erstellt...

```
RunspaceId      : 272cbf22-af71-494c-b1fe-215602195061
DisplayName      : Windows PowerShell Integrated Scripting Enviro...
Name            : PowerShell-ISE
Installed       : True
FeatureType     : Feature
Path            : Windows PowerShell Integrated Scripting Enviro...
Depth           : 1
DependsOn       : {NET-Framework-Core}
Parent          :
SubFeatures     : {}
SystemService   : {}
Notification    : {}
BestPracticesModelId :
AdditionalInfo   : {NumericId}
```

Hintergrund

Beim Exportieren von Befehlen aus einer aktiven PSSession generiert PowerShell automatisch ein Modul. Dieses Modul kann später von *Import-Module* wie jede andere PowerShell-Erweiterung nachgeladen werden.

Die Besonderheit von *Export-PSSession* ist, dass das Modul »weiß«, dass die Befehle eigentlich aus einer Remotesitzung stammen und lokal gar nicht vorhanden sind.

Wird das Modul später importiert und ein Cmdlet daraus lokal aufgerufen, regeneriert PowerShell die ursprüngliche PSSession und führt das Cmdlet mittels implizitem Remoting auf dem Remotesystem aus.

Natürlich kann das Modul nicht den gesamten Status einer vorherigen Remotesitzung einschließlich aller Änderungen und Anpassungen speichern, die darin vielleicht zuvor vorgenommen wurden. Deshalb müssen die Befehle, die Sie in das Modul exportiert haben, standardmäßig in der Remotesitzung vorhanden sein. Dies ist entweder der Fall, wenn Sie Cmdlets exportiert haben, die ohnehin zum Lieferumfang von Windows gehören, oder falls Sie eine *PSSessionConfiguration* eingerichtet haben, in der ein Autostart-Skript festgelegt ist, dass die Remotesitzung konfiguriert und beispielsweise weitere Befehle nachlädt.

Remoteauthentifizierung mit CredSSP einrichten

Sie möchten sich remote mit einem System verbinden und dort privilegierte Aufgaben durchführen, die es erforderlich machen, dass Ihre Anmeldedaten an Drittsysteme weitergereicht werden.

Lösung

Erlauben Sie zuerst dem Client, Anmeldeinformationen zu delegieren:

```
PS > Enable-WSManCredSSP -Role Client -DelegateComputer * -Force

cfg      : http://schemas.microsoft.com/wbem/wsmn/1/config/client/auth
lang     : de-DE
Basic    : true
Digest   : true
Kerberos : true
Negotiate : true
Certificate : true
CredSSP  : true
```

Möchten Sie die Delegation nur zu bestimmten Computern zulassen, ersetzen Sie den Stern durch die Liste der zugelassenen Computer.

ACHTUNG Befinden sich Client und Server nicht in derselben Domäne oder in überhaupt keiner Domäne, dann kann Kerberos als Anmeldeverfahren nicht verwendet werden und es muss auf NTLM-Authentifizierung umgestiegen werden. Weil es dabei keine gegenseitige Authentifizierung gibt, besteht das theoretische Risiko, dass Sie sich mit Anmeldeinformationen an einem bösartigen System ausweisen, das vorgibt, ein anderes System zu sein. Deshalb muss in einem solchen Fall noch per Gruppenrichtlinieneinstellung NTLM zugelassen werden. Sie finden die Einstellung beispielsweise in der lokalen Gruppenrichtlinie (*gpedit.msc*):

- *Computerkonfiguration/Administrative Vorlagen/System/Delegierung von Anmeldeinformationen/Delegierung von aktuellen Anmeldeinformationen mit reiner NTLM-Serverauthentifizierung zulassen*
- *Computer Configuration/Administrative Templates/System/Credentials Delegation/Allow Delegating Fresh Credentials with NTLM-only Server Authentication*

Aktivieren Sie die Richtlinie und fügen Sie der Liste der Richtlinie den Eintrag *wsman/** hinzu. Anstelle des Sterns darf auch ein spezifisches Computersystem genannt werden, wenn Sie die Weitergabe der Anmeldeinformationen nur an dieses System zulassen wollen.

Erlauben Sie dann dem Server, Anmeldeinformationen per *CredSSP* entgegenzunehmen. Die folgende Zeile muss dazu mit Administratorrechten auf dem Server ausgeführt werden. Entweder tun Sie dies vor Ort oder Sie verbinden sich per Remotedesktop mit dem System:

```
PS > Enable-WSManCredSSP -Role Server
```

Sobald CredSSP einmal eingerichtet ist, können Sie künftig Anmeldeinformationen weiterleiten, indem Sie den Parameter *-Authentication CredSSP* angeben:

```
PS > Enter-PSSession -computer storage1 -Authentication CredSSP
```

Hintergrund

Häufig müssen auf Remotesystemen Aufgaben durchgeführt werden, für die eine erneute Anmeldung erforderlich ist. Sie wollen sich beispielsweise mit einem Netzlaufwerk verbinden, für das Sie normalerweise ausreichende Berechtigungen besitzen, oder mit *Register-PSSession-Configuration* eine neue Remotekonfiguration einrichten.

Da Sie sich bereits an einem Remotesystem angemeldet haben, können Ihre aktuellen Anmeldeinformationen aus Sicherheitsgründen aber nicht mehr an Dritte weitergereicht werden und deshalb erhalten Sie auf Ressourcen und Befehle in der Remotesitzung keinen Zugriff, auf die Sie in einer lokalen Sitzung problemlos zugreifen können.

Entweder geben Sie in der Remotesitzung in diesem Fall erneut Ihre Anmeldeinformationen explizit an, verbinden sich also unter Angabe von Benutzernamen und Kennwort mit einem Netzlaufwerk. Oder Sie stellen zwischen Client und Server eine CredSSP-Vertrauensstellung her. In diesem Fall darf der Server Ihre Anmeldedaten an Dritte weiterreichen.

Als Beispiel, wann dies nützlich sein kann, soll eine Konfigurationsaufgabe dienen, deren Ergebnis im Abschnitt »Remotebefehle als Befehlerweiterung nutzen« ab Seite 523 benötigt wird. Auf einem Remoteserver vom Typ Windows Server 2008 R2 soll eine neue Remotekonfiguration namens *Serververwaltung* eingerichtet werden, die ein Startskript verwendet, mit dem das Modul *Servermanager* automatisch importiert wird.

Ohne CredSSP ist das remote nicht möglich:

```
PS > Enter-PSSession -ComputerName 192.168.2.234 -Cred powershell\Tobias
[192.168.2.234]: PS C:\Users\Tobias\Documents> Register-PSSessionConfiguration -Name
Serververwaltung -StartupScript $pshome\importservermanager.ps1
New-Item : Zugriff verweigert
Bei Zeile:18 Zeichen:17
```

Obwohl Sie Administrator des Zielsystems sind, wird der Zugriff verweigert, weil *Register-PSSessionConfiguration* eine erneute Anmeldung erfordert, Ihre Anmeldedaten aber in der Remotesitzung nicht weitergegeben werden. Mit *CredSSP* funktioniert die Aufgabe dagegen einwandfrei:

```
PS > Enter-PSSession -ComputerName 192.168.2.234 -Cred powershell\Tobias ->
-Authentication CredSSP
[192.168.2.234]: PS C:\Users\Tobias\Documents> Register-PSSessionConfiguration -Name
Serververwaltung -StartupScript $pshome\importservermanager.ps1

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Register-PSSessionConfiguration" für das Ziel "Name:
Serververwaltung. Erlaubt Administratoren die Remoteausführung von Windows PowerShell-
Befehlen auf diesem Computer.".
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [?] Hilfe
(Standard ist "J"):j

WSManConfig: Microsoft.WSMan.Management\WSMan::localhost\Plugin

Name                                Type                                Keys
----                                -
Serververwaltung                    Container                           {Name=Serververwaltung}

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "'Restart-Service'" für das Ziel "Name: WinRM".
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [?] Hilfe
(Standard ist "J"):j
```

Zwar kommt es anschließend zu einem Fehler, doch hat der nichts mit Authentifizierung zu tun, sondern eher damit, dass im Rahmen der Aufgabe der WinRM-Dienst neu gestartet wurde und daher Ihre Remotesitzung in sich zusammenbrach. Die Konfiguration war jedenfalls erfolgreich und die Fehlermeldung kann ignoriert werden.

Beim Ausführen von Daten für einen Remotebefehl ist folgender Fehler aufgetreten: Der Client kann keine Verbindung mit dem in der Anforderung angegebenen Ziel herstellen. Stellen Sie sicher, dass der Dienst auf dem Ziel ausgeführt wird und die Anforderungen akzeptiert. Lesen Sie die Protokolle und die Dokumentation für den WS-Verwaltungsdienst, der auf dem Ziel ausgeführt wird. Hierbei handelt es sich meistens um IIS oder WinRM. Wenn das Ziel der WinRM-Dienst ist, führen Sie den folgenden Befehl auf dem Ziel aus, um den WinRM-Dienst zu analysieren und zu konfigurieren: "winrm quickconfig". Weitere Informationen finden Sie im Hilfethema "about_Remote_Troubleshooting".

Ab sofort steht auf dem Server eine neue Konfiguration namens *Serververwaltung* zur Verfügung, die automatisch ein Skript unter *\$pshome\importservermanager.ps1* ausführt. Dieses Skript existiert allerdings noch gar nicht. Es soll das Modul *ServerManager* importieren. Um das Skript anzulegen, werden keine Anmeldedelegationen und also auch kein *CredSSP* mehr benötigt:

```
PS > Enter-PSSession -ComputerName 192.168.2.234 -Cred powershell\Tobias
[192.168.2.234]: PS C:\Users\Tobias\Documents> New-Item $pshome\importservermanager.ps1 -
type file -force -value 'Import-Module ServerManager'
```

```
Verzeichnis: C:\Windows\System32\WindowsPowerShell\v1.0
```

Mode	LastWriteTime	Length	Name
-a---	18.02.2011 16:45	27	importservermanager.ps1

```
[192.168.2.234]: PS C:\Users\Tobias\Documents> Set-ExecutionPolicy RemoteSigned -force
[192.168.2.234]: PS C:\Users\Tobias\Documents> Exit-PSSession
```

Nun ist die Konfiguration abgeschlossen. Wenn Sie sich künftig mit der Konfiguration Serververwaltung anmelden, stehen die Befehle des Moduls *ServerManager* automatisch zur Verfügung:

```
PS > Enter-PSSession -ComputerName 192.168.2.234 -Cred powershell\Tobias ->
-ConfigurationName Serververwaltung
[192.168.2.234]: PS C:\Users\Tobias\Documents> get-command -module server*
```

CommandType	Name	Definition
Cmdlet	Add-WindowsFeature	Add-WindowsFeature [-Name] <...
Cmdlet	Get-WindowsFeature	Get-WindowsFeature [[-Name] ...
Cmdlet	Remove-WindowsFeature	Remove-WindowsFeature [-Name...

```
[192.168.2.234]: PS C:\Users\Tobias\Documents> exit-ssession
```

Zusammenfassung

Remotenzugriffe spielen im administrativen Alltag eine enorme Rolle. Umso positiver ist es, dass zahlreiche Cmdlets wie beispielsweise *Get-Service* oder *Get-WmiObject* von sich aus Remotefähigkeiten mitbringen und Systeme abfragen können, die kein PowerShell ausführen.

Möglich ist dies, weil die meisten solcher Cmdlets hinter den Kulissen die alte DCOM-Technik für die Remotenzugriffe verwenden, also dieselbe Technik, die auch die Microsoft Management Console (MMC) einsetzt. Daraus erwachsen gleichzeitig die größten Probleme:

- Remoting funktioniert uneinheitlich. Jedes Cmdlet entscheidet selbst, ob es Remoting unterstützt, und falls ja, setzt es seine eigenen Remotingtechniken ein, deren Grundvoraussetzungen nicht immer klar dokumentiert sind. Entsprechend schwierig ist es, bei Remotingfehlern die Ursache zu finden.
- Die Netzwerkkonfiguration ist schwierig: Weil das klassische Remoting auf DCOM basiert, ist die Konfiguration der Firewall komplex und nicht selten ist die Remoteverbindung über DCOM überhaupt nicht möglich

PowerShell bietet deshalb ab Version 2.0 ein geradezu revolutionäres neues Konzept, das allerdings voraussetzt, dass sowohl Sender als auch Empfänger PowerShell 2.0 einsetzen. Dann aber

können Codestücke oder ganze Skripts auf einem oder Tausenden von Systemen gleichzeitig ausgeführt werden. Die Remotesysteme kommunizieren dabei über Webservices, die nur einen einzigen Firewallport benötigen, der darüber hinaus – wie auch die meisten anderen Aspekte des Remotings – frei konfigurierbar sind. Die unmittelbaren Vorteile sind also im Gegensatz zum klassischen Remoting:

- Remoting funktioniert absolut konsistent: Jeder beliebige Befehl, sogar native Konsolenbefehle, werden remotefähig, weil sie im Grunde weiterhin lokal ausgeführt werden, nur auf einem anderen System, und sich PowerShell – und nicht mehr der Befehl selbst – darum kümmert, wie die Ergebnisse zum Aufrufer zurückgelangen
- Die Netzwerkkonfiguration ist einfach und flexibel: PowerShell Remoting ist frei konfigurierbar und kann über eine Reihe von Protokollen und einen beliebigen Port kommunizieren

Wie erstaunlich die daraus resultierenden Möglichkeiten sind, zeigen einige Lösungen in diesem Kapitel: Mit implizitem Remoting können Befehle lokal ausgeführt werden, die eigentlich auf ganz anderen Systemen ablaufen. Innerhalb einer einzigen PowerShell-Pipeline könnten Daten von zig unterschiedlichen Servern gemeinsam bearbeitet werden.

Kapitel 19

Hintergrundjobs

In diesem Kapitel:

Cmdlets mit eigener Parallelverarbeitung finden	532
Aufgaben im Hintergrund ausführen	536
Aufgaben regelmäßig im Hintergrund ausführen	538
Anmeldeinformationen in Hintergrundjobs verwenden	540
Hintergrundjobs überwachen	541
Zusammenfassung	543

PowerShell kann stets nur eine Aufgabe nach der anderen abarbeiten. Um Aufgaben zu beschleunigen, lassen diese sich aber auch mithilfe von Hintergrundjobs parallel verarbeiten. Dabei wird eine Aufgabe an eine weitere (unsichtbare) PowerShell-Sitzung übergeben. Die Ergebnisse lassen sich von dort aus anschließend wieder abrufen.

Falls sich Ihre Aufgaben parallelisieren lassen, sind die möglichen Geschwindigkeitsgewinne dramatisch. So dauert das Überprüfen eines Netzwerksegments auf Computer, die online sind, in einem regulären Skript mehrere Minuten, weil alle Computer der Reihe nach getestet werden und es zu teils längeren Netzwerktimeouts kommen kann. Mithilfe der Hintergrundjobs lassen sich alle Computer gleichzeitig überprüfen und dieselbe Aufgabe wird in wenigen Sekunden erledigt.

Eine weitere Aufgabe der Hintergrundjobs ist die Ereignisbehandlung. Da Ereignisse jederzeit stattfinden können, muss ein Ereignishandler sofort darauf reagieren können. PowerShell kann bei Eintreffen von Ereignissen deshalb ebenfalls Hintergrundjobs starten, die das Ereignis verarbeiten.

Cmdlets mit eigener Parallelverarbeitung finden

Sie möchten wissen, welche Cmdlets über eine integrierte Parallelverarbeitung verfügen, Aufgaben also gleichzeitig und nicht nacheinander ausführen können.

Lösung

Listen Sie mit *Get-Help* alle Cmdlets auf, die den Parameter *-asJob* unterstützen:

```
PS > Get-Help * -parameter asJob
```

Name	Category	Synopsis
----	-----	-----
Invoke-Command	Cmdlet	Führt Befehle auf lokalen Comput...
Get-WmiObject	Cmdlet	Ruft Instanzen von WMI-Klassen (...)
Invoke-WmiMethod	Cmdlet	Ruft Methoden der Windows-Verwal...
Remove-WmiObject	Cmdlet	Löscht eine Instanz einer vorhan...
Set-WmiInstance	Cmdlet	Erstellt oder aktualisiert eine ...
Test-Connection	Cmdlet	Sendet ICMP-Echoanforderungspake...
Restart-Computer	Cmdlet	Startet das Betriebssystem auf d...
Stop-Computer	Cmdlet	Beendet lokale und Remotecompute...

Hintergrund

Einige Cmdlets beinhalten bereits die Fähigkeit, Aufgaben parallel zu bearbeiten, und unterstützen dann den Parameter *-asJob*. Wird er angegeben, führt das Cmdlet die Aufgabe im Hintergrund aus und gibt einen Hintergrundjob zurück. Über diesen Hintergrundjob lassen sich anschließend die Ergebnisse abrufen.

Cmdlets, die den Parameter *-asJob* unterstützen, setzen für die Parallelverarbeitung nicht ausschließlich zusätzliche PowerShell-Sessions ein. Es kann auch sein, dass das Cmdlet die Aufgabe intern über Multithreading löst. Möchten Sie zum Beispiel mit *Test-Connection* ein ganzes Netzwerksegment auf Computer überprüfen, die online sind, wäre es äußerst unökonomisch, für jeden Computer eine eigene separate PowerShell-Session zu öffnen.

Das folgende Beispiel zeigt, wie Sie durch die Parallelverarbeitung ein Netzwerksegment überprüfen:

```
PS > $ipliste = 1..255 | Foreach-Object { "192.168.2.$_" }

PS > Test-Connection -count 1 -ComputerName $ipliste -asJob |
>> Tee-Object -variable theJob |
>> Wait-Job |
>> Receive-Job |
>> Where-Object { $_.StatusCode -eq 0 } |
>> Select-Object -ExpandProperty Address
>>

PS > Remove-Job $theJob
```

Zuerst wird eine IP-Liste mit den IP-Adressen von *192.168.2.1* bis *192.168.2.255* generiert. Danach wird diese Liste an *Test-Connection* übergeben und das Cmdlet mit *-asJob* beauftragt, die Computer gleichzeitig zu überprüfen. Das Ergebnis ist ein Hintergrundjob.

Dieser wird zuerst an *Tee-Object* weitergeleitet, wo der Job in der Variablen *\$theJob* vermerkt wird, um ihn später wieder entfernen zu können. Danach wird er an *Wait-Job* geleitet. PowerShell wartet nun, bis der Job erledigt ist.

HINWEIS

Es klingt auf den ersten Blick etwas seltsam, einen Hintergrundjob zu erzeugen und danach auf ihn zu warten. Hätte man die Aufgabe dann nicht gleich ohne Hintergrundjob erledigen können? Der fundamentale Unterschied ist, dass *Test-Connection* mit *-asJob* die Aufgabe auf viele verschiedene Threads aufteilt, die alle gleichzeitig laufen. *Wait-Job* synchronisiert das Skript also, indem es wartet, bis alle parallel laufenden Arbeiten abgeschlossen sind.

Sobald der Job abgeschlossen ist, empfängt *Receive-Job* die Ergebnisse und *Where-Object* wählt nur die Ergebnisse aus, bei denen die Eigenschaft *Status* den Wert 0 enthält, der Computer also geantwortet hat und online ist. *Select-Object* gibt den Namen des Computers zurück. Nun kann der Hintergrundjob mit *Remove-Job* wieder entfernt werden.

Erreichbare Computer im Netzwerk zu finden, ist eine wichtige Aufgabe, und deshalb kann man den Code von eben in eine praktische Funktion verwandeln:

```
Function Test-Online {
    Param(
        [String[]]
        $computername
    )
    Test-Connection -count 1 -ComputerName $computername -asJob |
    Tee-Object -variable theJob |
    Wait-Job |
    Receive-Job |
    Where-Object { $_.StatusCode -eq 0 } |
    Select-Object -ExpandProperty Address |
    Sort-Object { [System.Version]$_ }

    Remove-Job $theJob
}
```

TIPP

Die Funktion *Test-Online* liefert die IP-Adressen der gefundenen Systeme als sortierte Liste zurück. Dazu werden die Ergebnisse zum Schluss der Pipeline an dieses Cmdlet weitergeleitet:

```
| Sort-Object { [System.Version]$_ }
```

Da die IP-Adressen als Text zurückgeliefert werden, kann man sie normalerweise nur alphanumerisch sortieren. Damit die IP-Adressen korrekt sortiert werden, verwendet *Sort-Object* als Sortierkriterium deshalb nicht die IP-Adresse selbst, sondern wandelt sie zuerst in den Typ *[System.Version]* um. Dieser Typ ist eigentlich dazu gedacht, Versionsnummern zu repräsentieren, aber weil Versionsnummern genau wie IP-Adressen aus vier separaten Zahlen bestehen, kann man über diesen Kniff auch IP-Adressen korrekt sortieren.

Mit ihrer Hilfe könnten Sie nun beispielsweise DNS-Einträge überprüfen:

```
PS > $ipliste = 1..255 | Foreach-Object { "192.168.2.$_" }
PS > $online = Test-Online $ipliste
PS > $online | Foreach-Object { $ip = $_; try { [System.Net.DNS] ::GetHostByAddress($ip) } catch { Write-Warning "$($ip): → $($_.Exception.InnerException.Message)"; $global:t=$_ } }
```

HostName	Aliases	AddressList
speedport.ip	{}	{192.168.2.1}
WARNUNG: 192.168.2.100: Der angeforderte Name ist gültig, es wurden jedoch keine Daten des angeforderten Typs gefunden		
demo5.Speedport_W_700V	{}	{192.168.2.101}
demo5.Speedport_W_700V	{}	{192.168.2.105}
storage1	{}	{192.168.2.108}
WIN-KHHRLLOTMS3	{}	{192.168.2.234}

Oder Sie könnten per WMI Informationen über Remotesysteme abfragen, beispielsweise deren BIOS-Informationen:

```
PS > $online | Foreach-Object { Get-WMIObject Win32_BIOS -computername $_ }
```

Hier würden Sie allerdings noch Fehler erhalten, falls die Firewall des Systems Ihren Zugriff nicht erlaubt oder Sie nicht über ausreichende Berechtigungen verfügen. Deshalb sollte man diese Fehler abfangen:

```
$online | Foreach-Object {
    $ip = $_
    try { Get-WMIObject Win32_BIOS -computername $ip -ea Stop | →
        Select-Object Manufacturer, SerialNumber, __Server }
    catch [System.UnauthorizedAccessException] →
        { Write-Warning ('{0,-20}{1,-40}' -f $ip, →
            'fehlende Zugriffsrechte') }
    catch {
        if ($_.Exception.Message -like '*RPC*') {
            Write-Warning ('{0,-20}{1,-40}' -f $ip, →
                'Firewall verhindert Kontakt oder System offline')
        } else {
            Write-Warning ('{0,-20}{1,-40}' -f $ip, $_.Exception.Message)
        }
    }
}
```

Das Ergebnis könnte nun so aussehen:

Manufacturer	SerialNumber	__SERVER
-----	-----	-----
Phoenix Technologies Ltd.	ZAMA93HS600210	DEM05
WARNING: 192.168.2.1	Firewall verhindert Kontakt oder System offline	
WARNING: 192.168.2.108	fehlende Zugriffsrechte	
WARNING: 192.168.2.100	Firewall verhindert Kontakt oder System offline	
WARNING: 192.168.2.103	Firewall verhindert Kontakt oder System offline	
WARNING: 192.168.2.234	fehlende Zugriffsrechte	

HINWEIS

Warum könnte man nicht *Get-WmiObject* ebenfalls als Hintergrundjob ausführen lassen? Beispielsweise so:

```
PS > Get-WmiObject Win32_BIOS -computername $online -asJob |
>> Tee-Object -variable theJob |
>> Wait-Job |
>> Receive-Job
>>

PS > Remove-Job $theJob
```

Prinzipiell ist das möglich, doch bestimmt hierbei jedes Cmdlet selbst, wie es auf Fehler reagiert. Bei *Get-WmiObject* wird die gesamte Parallelverarbeitung sofort abgebrochen, wenn ein schwerwiegender Fehler auftritt. Als solcher gilt (leider) auch, wenn eines der Systeme nicht antwortet, beispielsweise weil seine Firewall den Zugriff nicht erlaubt. Sie können die Parallelverarbeitung hier also nur dann einsetzen, wenn Sie sicher sind, dass alle angegebenen Systeme fehlerfrei abgefragt werden können.

Aufgaben im Hintergrund ausführen

Sie möchten eine Aufgabe, die längere Zeit in Anspruch nimmt, im Hintergrund ausführen. Sie wollen entweder ungestört an anderen Dingen weiterarbeiten, während die Aufgabe bearbeitet wird. Oder Sie wollen mehrere Aufgaben im Hintergrund parallel ausführen, um die Geschwindigkeit Ihres Skripts zu erhöhen.

Lösung

Starten Sie mit *Start-Job* einen Hintergrundjob und übergeben Sie diesem den Code, der ausgeführt werden soll. Die folgende Zeile sucht rekursiv alle *.log*-Dateien aus dem Windows-Ordner, was längere Zeit in Anspruch nimmt. Der Hintergrundjob führt diese Aufgabe in einer separaten unsichtbaren Sitzung aus und liefert nur ein Jobobjekt zurück:

```
PS> $job = Start-Job { Dir $env:windir *.log -Recurse →
    -EA SilentlyContinue }
```

Mit *Get-Job* prüfen Sie, ob der Hintergrundjob seine Arbeit erledigt hat:

```
PS> $job | Get-Job
```

Id	Name	State	HasMoreData	Location	Command
1	Job1	Running	True	localhost	Dir \$env:...

Meldet ein Job in seiner Spalte *State* den Zustand *Running*, läuft die Aufgabe noch. Steht in *HasMoreData* ein *True*, sind bereits Ergebnisse abrufbereit. Ist der Job abgeschlossen, meldet *Get-Job* in *State* den Zustand *Completed*:

Die Ergebnisse des Jobs rufen Sie dann mit *Receive-Job* ab. Geben Sie den Parameter *-Keep* nicht an, werden die Ergebnisse, die Sie abrufen, aus dem Job entfernt.

```
PS> Receive-Job $job
```

Verzeichnis: C:\Windows

Mode	LastWriteTime	Length	Name
-a---	16.11.2009 11:30	6518	DPINST.LOG
-a---	01.11.2009 09:21	1774	DtcInstall.log
-a---	04.11.2009 12:28	864	PFR0.log
-a---	04.03.2010 15:03	26744	setupact.log
(...)			

Der Job wird zum Schluss mit *Remove-Job* entfernt:

```
PS> Remove-Job $job
```


Möglich ist das allerdings erst, wenn der Hintergrundjob abgeschlossen ist. Mit *Stop-Job* brechen Sie Hintergrundjobs vorzeitig ab. Sie können auch den Parameter *-Force* mit *Remove-Job* verwenden.

Hintergrund

Möchten Sie mehrere Aufgaben parallelisieren, starten Sie mehrere Jobs, die dann alle gemeinsam ausgeführt werden. Mit *Wait-Job* können Sie diese synchronisieren, also Ihr Skript so lange pausieren lassen, bis alle Ergebnisse vorliegen.

Das folgende Skript führt drei Aufgaben durch. *Measure-Command* misst die Ausführungsgeschwindigkeit:

```
PS > Measure-Command {  
>> $dlls = Dir $env:windir\system32\*.dll | Select-Object -expand VersionInfo  
>> $hotfixes = Get-Hotfix  
>> $processes = Get-Process  
>> }  
  
>>
```

Um diese drei Aufgaben zu parallelisieren, werden sie jeweils mit *Start-Job* in Hintergrundjobs ausgeführt, die dann mit *Wait-Job* synchronisiert werden. Auch hier wird die Ausführungsgeschwindigkeit gemessen.

```
PS > Measure-command {  
>> $job1 = Start-Job { Dir $env:windir\system32\*.dll | →  
    Select-Object -expand VersionInfo }  
>> $job2 = Start-Job { Get-Hotfix }  
>> $job3 = Start-Job { Get-Process }  
>> Wait-Job $job1, $job2, $job3 | Out-Null  
>> $dlls = Receive-Job $job1  
>> $hotfixes = Receive-Job $job2  
>> $processes = Receive-Job $job3  
>> Remove-Job $job1, $job2, $job3  
>> }  
>>
```

Für zuverlässige Messergebnisse lassen Sie beide Messungen mehrmals durchführen. Das Lesen der Dateiversioneninformationen dauert beispielsweise beim ersten Mal länger, weil das Dateisystem die Informationen anschließend puffert.

Wie sich herausstellt, sind Hintergrundjobs keine Garantie für mehr Geschwindigkeit und können ein Skript sogar verlangsamen. Erstens besitzt jeder Hintergrundjob einen gewissen Overhead und zweitens müssen die Ergebnisse vom Hintergrundjob über XML-Serialisierung an Ihre Sitzung zurückgeliefert werden. Deshalb sollten nur »lohnenswerte« Aufgaben als Hintergrundjob ausgeführt werden.

Lohnenswert sind Aufgaben, wenn sie sehr lange dauern, aber möglichst wenige Ergebnisse zurückliefern (weil diese durch die Serialisierung zu Verzögerungen führen, die den Geschwindigkeitsgewinn wieder auffressen können).

Das Abrufen der Hotfixes und Prozesse waren deshalb keine lohnenswerten Aufgaben für Hintergrundjobs. Das Abrufen von Dateiversionen war eine zumindest bedingt lohnenswerte Aufgabe, weil sie zwar längere Zeit benötigt, aber andererseits viele Ergebnisse zurückliefert.

Hintergrundjobs verwenden dasselbe Sessionmodell wie PowerShell Remoting, sodass hier ähnliche Einschränkungen gelten: die Ergebnisse, die Sie von Hintergrundjobs empfangen, werden über XML-Serialisierung übertragen. Die empfangenen Objekte können also nur noch gelesen, aber nicht verändert werden und enthalten auch keine Methoden mehr.

Aufgaben regelmäßig im Hintergrund ausführen

Sie möchten eine bestimmte Aufgabe immer wieder in Intervallen ausführen. Sie wollen zum Beispiel prüfen, ob ein kritischer Server online ist, oder Sie möchten die Titelleiste der Konsole in regelmäßigen Intervallen mit den neuesten Schlagzeilen aus dem Internet füllen.

Lösung

Verwenden Sie die beiden folgenden Funktionen *New-IntervalJob* und *Remove-IntervalJob*, um Aufgaben in regelmäßigen Intervallen auszuführen

```
function New-IntervalJob {
    param(
        [int]
        $seconds,
        [ScriptBlock]
        $action,
        $id='Default'
    )
    $timer = New-Object System.Timers.Timer
    $timer.Interval = $seconds * 1000
    $timer.Enabled = $true
    Register-ObjectEvent $timer "Elapsed" -SourceIdentifier $id →
        -Action $action
}

function Remove-IntervalJob {
    param(
        $id = 'Default'
    )
    Unregister-Event $id
    Remove-Job -name $id
}
```

Der folgende Code gibt alle zehn Sekunden eine Meldung mit einem Piepton in die Konsole aus:

```
PS > $code = { Write-Host "Hintergrundjob`a" }
PS > New-IntervalJob -id test $code -sec 10
```

Um den Hintergrundjob wieder zu entfernen, verwenden Sie *Remove-IntervalJob*:

```
PS > Remove-IntervalJob test
```

Hintergrund

Hintergrundjobs lassen sich nicht nur dazu verwenden, Aufgaben im Hintergrund zu bearbeiten. Man kann Hintergrundjobs auch dazu einsetzen, um auf »Ereignisse« zu reagieren. *New-IntervalJob* legt dazu ein *Timer*-Objekt an. Dieses Objekt feuert in regelmäßigen Abständen Ereignisse ab.

Welche Ereignisse ein Objekt abfeuern kann, verrät *Get-Member*:

```
PS > $timer = New-Object system.timers.timer
PS > $timer | Get-Member -MemberType Event

    TypeName: System.Timers.Timer

Name      MemberType Definition
----      -
Disposed  Event      System.EventHandler Disposed(System.Object, System...
Elapsed   Event      System.Timers.ElapsedEventHandler Elapsed(System...
```

Register-ObjectEvent beauftragt PowerShell, auf Ereignisse zu reagieren und einen Skriptblock im Hintergrund auszuführen. Hierbei werden also automatisch Hintergrundjobs erzeugt. Die Ereignisbearbeitung wird so lange fortgesetzt, bis mit *Unregister-Event* der Eventhandler und der automatisch generierte Hintergrundjob wieder entfernt werden.

Das Beispiel zeigt also gleich zwei Möglichkeiten der PowerShell:

- **Regelmäßig Aufgaben ausführen** Sie können mithilfe des *Timer*-Objekts regelmäßig auftretende Ereignisse generieren und so Hintergrundjobs in regelmäßigen Abständen automatisch ausführen
- **Objektereignisse verarbeiten** Sie können aber auch ganz andere Objektereignisse verarbeiten und mithilfe von Hintergrundjobs darauf reagieren

Der folgende Skriptblock lädt alle fünf Minuten die aktuellen RSS-Schlagzeilen des Heise-News-tickers herunter und schreibt jeweils eine zufällige Schlagzeile in die Titelleiste der Konsole:

```
$code = {  
    if (!(Test-Path variable:newsticker)) {  
        $newsticker = New-Object XML  
        $newsticker = $newsticker | Add-Member NoteProperty LastUpdate →  
            (Get-Date -Year 2000) -PassThru  
    }  
  
    if ($newsticker.lastupdate -lt (Get-Date).AddMinutes(-5)) {  
        Write-Progress "Refreshing Ticker" "Refresh"  
        $newsticker.Load("http://www.heise.de/newsticker/heise-atom.xml")  
        $newsticker.LastUpdate = Get-Date  
        Write-Progress -Completed "Refreshing Ticker" "Refresh"  
    }  
  
    $entry = $newsticker.feed.entry | Get-Random  
    $Host.UI.RawUI.WindowTitle = $entry.title  
}
```

Sie sollten den Code zunächst testen. Rufen Sie den Code auf, sollte eine neue Heise-Schlagzeile in der Titelleiste der Konsole erscheinen:

```
PS > . $code
```

Falls dies nicht geschieht, verfügen Sie vermutlich entweder nicht über eine Internetverbindung oder Sie können nur über einen Proxy mit separater Authentifizierung auf das Internet zugreifen. In diesem Fall kann das XML-Objekt dieses Beispiels die Informationen nicht aus dem Internet abrufen.

Wurde die Schlagzeile korrekt geladen, kann die Aufgabe jetzt regelmäßig ausgeführt werden. Die nächste Zeile liefert alle 30 Sekunden eine neue Schlagzeile in die Titelleiste der PowerShell-Konsole:

```
PS > New-IntervalJob -id ticker $code -sec 30
```

Anmeldeinformationen in Hintergrundjobs verwenden

Sie möchten aus einem Hintergrundjob heraus auf kennwortgeschützte Ressourcen zugreifen.

Lösung

Erfragen Sie die Anmeldeinformationen im Hauptcode und übergeben Sie die Anmeldeinformationen dann als Parameter an den Hintergrundjob.

Die folgende Zeile erfragt zuerst Anmeldeinformationen und übergibt diese dann mit `-ArgumentList` an den Code des Hintergrundjobs. Dieser kann sich dann mit den Anmeldeinformationen an einem weiteren System ausweisen und per WMI BIOS-Informationen abfragen:

```
PS > Start-Job { param($cred) Get-WmiObject Win32 BIOS -ComputerName storage1 -Credential $cred } -ArgumentList (Get-Credential)
```

Hintergrund

Sie dürfen Anmeldeinformationen in einem Hintergrundjob nicht interaktiv mit *Get-Credential* erfragen, weil dies den Hintergrundjob blockieren würde. Der Hintergrundjob läuft schließlich unsichtbar in einer anderen Session, sodass Sie das Dialogfeld für die Kennworteingabe nicht sehen könnten. Erst wenn Sie die Ergebnisse des blockierten Jobs mit *Receive-Job* empfangen, erscheint das Dialogfeld. Keine gute Idee.

Besser ist, Anmeldeinformationen über Parameter und `-ArgumentList` an den Code des Hintergrundjobs weiterzureichen. Sie können die Anmeldeinformation alternativ auch hardcodiert angeben:

```
PS > Start-Job { Get-WMIObject Win32 BIOS -Computer storage1 -Credential (New-Object System.Management.Automation.PSCredential('Administrator', ('topSecret99' | ConvertTo-SecureString -Force -asPlainText))) }
```

Hintergrundjobs überwachen

Sie möchten alarmiert werden, wenn ein Hintergrundjob seine Arbeit erledigt hat.

Lösung

Möchten Sie nur auf einen oder auf mehrere Hintergrundjobs warten, verwenden Sie *Wait-Job* und geben die Hintergrundjobs an, auf die Sie warten möchten.

```
PS > $job1 = Start-Job { Get-Process }
PS > $job2 = Start-Job { Get-Hotfix }
PS > Wait-Job $job1, $job2
PS > Receive-Job $job1, $job2
PS > Remove-Job $job1, $job2
```

Wollen Sie eine Meldung erhalten, sobald ein Job beendet wurde, verwenden Sie die folgende Funktion *Start-MonitoredJob*:

```
function Start-MonitoredJob {
    param(
        [ScriptBlock]
        $code,
        $id = 'default',
        $variable = 'ergebnis'
    )

    $action = {
        if($sender.State -eq 'Completed') {
            $variable = $event.MessageData
            Write-Host ('Hintergrundjob "{0}" ist fertig!' -f →
                $event.SourceIdentifier) -Back 'White' -Fore 'Red'
            Write-Host ('Ergebnis liegt in ${0} vor.' -f $variable) →
                -Back 'White' -Fore 'Red'
            Write-Host (prompt) -NoNewline

            Set-Variable $variable -Scope global -Value (Receive-Job $sender)
            Unregister-Event -SourceIdentifier $event.SourceIdentifier
            Remove-Job -Name $event.SourceIdentifier
            Remove-Job $sender
        } else {
            Write-Host ('Job "{0}" hat Status geändert: {1}' -f →
                $event.SourceIdentifier, $sender.State) -Back 'White' →
                -Fore 'Red'
            Write-Host (prompt) -NoNewline
        }
    }

    $job = Start-Job $code
    Register-ObjectEvent $job -EventName StateChanged -SourceIdentifier →
        $id -MessageData $variable -Action $action
}
```

Wenn Sie auf diese Weise einen Hintergrundjob starten, wird intern eine Überwachung eingerichtet. Ist der Job abgeschlossen, werden die Ergebnisse automatisch in eine von Ihnen angegebene Variable gespeichert und der Job entfernt:

```
PS > Start-MonitoredJob -code { Get-Hotfix } -id hotfixe -variable liste | →
    Out-Null
```

Ist der Job fertig, erhalten Sie eine Meldung wie diese:

```
PS > Hintergrundjob "hotfixe" ist fertig!
Ergebnis liegt in $liste vor.
PS > $liste
```

Sie können nun die Ergebnisse in *\$liste* abrufen oder weiterverarbeiten.

Hintergrund

In diesem Beispiel wird mit *Register-ObjectEvent* das Ereignis *StateChanged* des Jobs überwacht. Ändert sich der Status zu »Completed«, wird die Überwachung beendet und die Ergebnisse des Jobs abgerufen.

Das Beispiel zeigt darüber hinaus, wie Informationen zwischen dem Hintergrundjob und der interaktiven Sitzung ausgetauscht werden. Das ist nicht ganz trivial, da es sich um zwei voneinander getrennte PowerShell-Sessions handelt.

- **\$event.SourceIdentifier** Liefert die ID der Überwachung. Auf diese Weise kann sich die Überwachung mit *Unregister-Event* und *Remove-Job* wieder selbst entfernen.
- **\$sender** Enthält das Jobobjekt, das das Ereignis ausgelöst hat. So kann man die Ergebnisse des Jobs abrufen und den Arbeitsjob danach entfernen.
- **\$event.MessageData** Ruft zusätzliche Informationen ab, die beim Aufruf von *Register-ObjectEvent* mit *-MessageData* übergeben wurden. Im Beispiel wird auf diese Weise der Name der Variable übergeben, in der später die Ergebnisse des Hintergrundjobs gespeichert werden sollen. Die Variable selbst wird mit *Set-Variable* im globalen Kontext angelegt, damit sie auch außerhalb des Eventhandlers zur Verfügung steht.

Zusammenfassung

Mit Hintergrundjobs kann man parallelisierbare Aufgaben beschleunigen. Besonders geeignete Cmdlets bieten dafür den Parameter *-asJob*. Eigener Code kann mit *Start-Job* als Hintergrundjob gestartet werden. Sobald Sie einen Hintergrundjob starten, liegt es in Ihrer Verantwortung, den Status des Hintergrundjobs zu überwachen und die Ergebnisse abzurufen.

Zuständig dafür sind die Cmdlets aus der Familie *Job*:

```
PS > Get-Command -Noun Job
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Get-Job	Get-Job [[-Id] <Int32[]>] [-Verbose] [-Debug]...
Cmdlet	Receive-Job	Receive-Job [-Job] <Job[]> [[-Location] <Stri...
Cmdlet	Remove-Job	Remove-Job [-Id] <Int32[]> [-Force] [-Verbose...
Cmdlet	Start-Job	Start-Job [-ScriptBlock] <ScriptBlock> [[-Ini...
Cmdlet	Stop-Job	Stop-Job [-Id] <Int32[]> [-PassThru] [-Verbos...
Cmdlet	Wait-Job	Wait-Job [-Id] <Int32[]> [-Any] [-Timeout <In...

PowerShell generiert mitunter automatisch Hintergrundjobs, zum Beispiel, wenn Sie Objekt-ereignisse überwachen. Weil Ereignisse jederzeit auftreten können, wird in diesem Fall ein eigener Hintergrundjob für die Ereignisüberwachung eingerichtet.

Stichwortverzeichnis

- .cer-Datei 391
- .pfx-Datei 402
- .pvk-Datei 391
- #text 427
- %path% 299
- \$DebugPreference 271
- \$error 198–199
- \$ErrorActionPreference 199
- \$event.MessageData 543
- \$event.SourceIdentifier 543
- \$Foreach.Current 166–167
- \$Foreach.MoveNext 167
- \$home 209, 221
- \$Host.PrivateData 199
- \$Input 140, 178
- \$LastExitCode 181, 198–199, 309
- \$Matches 46, 67, 158
- \$Null 125, 298, 325
- \$OFS 102
- \$profile 177, 221
- \$pshome 221
- \$PSScriptRoot 503
- \$PSSessionConfigurationName 518
- \$pwd 221
- \$sender 543
- \$true 146
- 32-Bit-Umgebung 430–431
 - PowerShell-Konsole 430
- 64-Bit-Umgebung 430

A

- Abgeschnittene Informationen 63
- Access 365, 434
 - Datenbank 431
 - Version 2007 435
- Access Control Entry-Typen 377
- AccessControlType 365, 367
- AccountExpirationDate 471
- Active Directory 434
 - anmelden 456
- Active Directory Service Interface 477
- ActiveDirectory 500
- Add 471

- Add-ADGroupMember 475
- Add-Content 230
 - encoding 230
- AddDays 89
- AddHours 89
- Add-LocalGroup 485
- Add-Member 296, 330
- AddMilliseconds 90
- AddMinutes 90
- AddMonths 90
- AddNew 446
- Add-PSSnapin 496
- AddSeconds 90
- AddTicks 90, 92
- AddYears 90
- adLockBatchOptimistic 448
- adLockOptimistic 448
- adLockPessimistic 448
- adLockReadOnly 448
- adLockUnspecified 448
- ADO.NET 430, 446
- ADODB-Connection 445
- adOpenDynamic 448
- adOpenForwardOnly 448
- adOpenKeyset 448
- adOpenStatic 448
- adOpenUnspecified 448
- After 339
- Alias
 - Definitionen exportieren und importieren 308
 - einrichten 307
 - löschen 309
- Aliasnamen-Parameter 180
- AllSigned 175, 384
- AllUsersProfile 222
- And 151
- Anführungszeichen 18
 - einfache, doppelte 20
 - im Text darstellen 28
 - in Text einfügen 19
- Anker 44
- Anmeldeinformationen
 - delegieren 525
 - in Skript angeben 302

Anweisung ausführen 300
 Anwendungs- und Dienstprotokolle 339
 Anwendungsdaten 223
 AppData 222
 Append 63
 AppendChild 427
 ApplicationData 223
 ArgumentList 516, 541
 Array *siehe* Arrays
 ArrayList 99, 110
 Arrays
 assoziative 115
 Auf Element prüfen 107
 auf Element zugreifen 101
 Elemente hinzufügen oder entfernen 109
 Elemente in Schleife bearbeiten 103
 Gesamtgröße 101
 Index 101
 mehrdimensionale 100
 neu anlegen 96
 sortieren 106
 Typ prüfen 147
 Typisierung 99
 zusammenfassen 112
 as 86
 AS/400 436
 ASCII-Codetabelle 162
 asHash 140
 asJob 532–533
 asString 140
 Attrib.exe 240–241
 Ausführbare Anweisung 300
 Authentication 526
 AutoSize 129
 Autostart-Programmgruppe 223

B

Backtick-Zeichen 19, 28–29, 174
 Backup
 Ereignisprotokoll 354
 BackupEventLog 354
 BaseStream 259
 Bedingungen 153
 Before 339
 Begin 178
 Block 133, 177
 Benutzerkonto 473, 476
 deaktivieren, lokales 489
 Berechtigungseinträge 366
 Besitz übernehmen 377
 Besitzer eines Objekts 365
 Bilder-Ordner 223
 Bildschirmauflösung 128
 Binärdaten 231
 Binary 269
 Bitmaske 243, 366
 BitsTransfer 495
 Bool 86
 Break 199
 Byte-Array 59, 231

C

Cache 434
 cacls.exe 364, 368, 374
 CaseSensitive 134
 catch-Block 198
 CategoryInfo 199
 Ccontains 149
 cd-Befehl 208, 246
 Ceiling 416
 ceq 149
 cert 406
 cge 149
 cgt 149
 Change 324
 ChangePassword() 463
 ChangePasswordAtLogon 459
 ChangeStartMode 324
 Char 59
 Check-Date 86
 cle 149
 Clear 471
 Clear-Content 233
 Clear-ItemProperty 282, 284
 clike 40, 125
 clt 149
 cmatch 45
 cmd.exe 301
 cne 149
 cnotcontains 149
 Codesignatur 385
 CodeSigningCert 384–385
 Codesigning-Zertifikat 384, 397
 erstellen 389
 command 305
 CommandLine 329
 CommitChanges 485
 CommonApplicationData 223
 CommonProgramFiles 222–223
 COMObject 237, 445

- Compare-Object 113, 251, 332
 - property 332
- Computername 508, 532
- ConfigurationName 518
- confirm 235, 283, 462
- contains 40, 107–108, 149, 152, 279, 415
- ContainsWildcardCharacter 217
- continue 204
- ConvertToDateTime 90, 327
- ConvertTo-XML 425
- Cookies 223
- copy 224
- Copy-Item 224
 - recurse 225
- count 415, 441
- CreateCommand 440
- CreateConnection 439
- Create-EncryptedCredentialScript 302
- CreateTrust 391
- CreationTime 89, 327
- Credential 508
 - in Skript angeben 302
- CredSSP 525–526
- CredSSP-Vertrauensstellung 526
- CurrentDirectory 216, 355
- Cursor (Datenbank) 447

D

- Database 433, 435
- Dataset-Objekt 441
- Datasource 435
- Date 78
- Datei
 - Änderungen überwachen 252
 - Attribute 240
 - Attribute ändern 240
 - Attribute lesen 240
 - Inhalt lesen 24, 27, 232
 - Inhalt löschen 233
 - kopieren 224
 - löschen 234
 - Metadaten lesen 237
 - mit Attribut finden 242
 - neu anlegen 227
 - schreiben 230
 - Suche 211
 - Text ohne Zeilenumbruch anfügen 63
 - umbenennen 235
 - Zugriffsrechte 364
- Dateierweiterung 45
 - ermitteln 46
 - finden 49
- Dateiname, temporärer 79
- Dateisystem, Platzhalterzeichen 244
- Datenbankformate 430
- Datenbankprovider 430
- DateTime 73, 79, 85, 91
- Datum
 - aktuelles 72
 - Systemzeit 72
 - WMI-Format 90
 - Zeitdifferenz 87
- Datum und Zeit
 - formatieren 36
- Day 78
- DayOfWeek 78
- DayOfYear 78
- Dbq 433
- Default 167, 382, 384
- Deinstallationsvorgänge 339
- del *siehe* Remove-Item
- Delegation 525
- Delete 332, 441, 491
- DeleteValue 285
- DependentServices 323
- descending 134–135
- Description 467
- Desktop 223
- DesktopDirectory 223
- Desktop-Ordner 223
- Dictionary 115
- Dienst 316
 - abhängiger 323
 - anhalten 321
 - Anmeldekonto 324
 - Auf Status warten 326
 - auflisten 316
 - Einstellungen ändern 324
 - entfernen 332
 - EXE-Datei finden 329
 - finden 318
 - fortsetzen 321
 - Kennwort ändern 325
 - Laufzeit bestimmen 327
 - neu installieren 332
 - Prozess-ID 329
 - starten 321
 - stoppen 321
 - Uptime 327
- dir *siehe* Get-ChildItem
- Direktvariablen 21, 32

Disable-PSRemoting 511
 div 416
 do 160
 Dokumente-Ordner 223
 DoNotOverwrite 357
 Driver 433
 DWord 270

E

ea 298
 Editor 172
 Einwählerlaubnis 467
 Else 155
 ElseIf 155
 E-Mail-Adresse überprüfen 46
 Enable-ADAccount 459
 Enable-PSRemoting 510
 Encoding 63, 230
 Byte 231
 Probleme 64
 End 178
 End-Block 133
 end-Block 178
 EndsWith 41
 Enter-PSSession 512
 EntryType 349–350
 Environment 221–222, 355
 eq 108, 149
 Ereignisanzeige
 abweichende Angaben 344
 Ereignis-ID 343
 Ereignisprotokolle 336
 archivieren 354
 auswerten 64
 Backup 354
 bestimmte Ereignis-IDs 342
 EntryType 349
 Ereignishäufigkeit 351
 Ereignisse einer bestimmten Quelle 343
 Ereignisstichwort finden 350
 Fehler finden 349
 konfigurieren 358
 löschen 357
 Maximalgröße 358
 MaximumKilobytes 337
 MinimumRetentionDays 360
 neuen Eintrag schreiben 353
 neueste Einträge 338
 OverflowAction 337
 prüfen, ob vorhanden 338
 Remotezugriff 357
 sichern 354
 Überschreibmodus 358
 WMI 344
 Ereignisquelle 343, 353
 neu anlegen 353
 vorhandene anzeigen 354
 Error Record 199
 ErrorAction 199, 298
 ErrorBackgroundColor 199
 ErrorDetails 199
 Error-Ereignisse 349
 ErrorForegroundColor 199
 ErrorLevel 180, 198, 311
 Ersetzen von Text 51
 Escape() 48
 Escape-Zeichen 245
 ETS 127
 EventData 347
 EventID 352
 examples 191
 Excel 330, 434, 436
 Exception 199
 ExecuteNonQuery 441
 ExecuteReader 440–441, 445
 ExecuteRow 441
 ExecuteScalar 441
 ExecuteXmlReader 441, 445
 ExecutionPolicy 175, 396
 Exists 338
 Exit 180
 Exit-PSSession 512
 ExpandString 269
 Export 401–402
 Exportable 388
 Export-Alias 308
 Export-Clixml 114, 331, 355, 418, 423
 Export-Csv 330
 Export-ModuleMember 504
 Export-PSSession 523–524
 Export-Xml 422–423
 eXtended Markup Language 408
 Extended Type System 127

F

Favorites 223
 Fehlermeldungen
 Farbe ändern 199
 Farbe festlegen 199
 FieldCount 440

- FileSystemRights 365, 367
- FileSystemWatcher 252
- Filter 196
- Fingerabdruck 391
- Firewall 511
- Following-Sibling 416
- For 161–162
- FOR XML 443
- ForEach-Object 130, 164
- ForEach-Schleife 103, 164, 422
- Format 79
- format default 474
- Formatierungsoperator 33
- Formatierungsoptionen 36
- Format-List 129, 417
- Format-Table 129, 330
 - Wrap 339
- formatType 501
- f-Parameter 21, 23, 33–34, 79
- FromFileTime 91
- FromFileTimeUTC 91
- FullName 217
- FullyQualifiedErrorID 199
- Function 192
- FunctionsToExport 504
- Funktionen 192, 308
 - Hilfe hinzufügen 190

G

- Gerätetreiberdienste 320
- Get 472
- Get-Acl 364–365
- Get-ADComputer 467
- Get-ADGroup 467
- Get-ADOrganizationalUnit 468
- Get-ADUser 465, 469
- Get-Alias 308
- Get-AuthenticodeSignature 399–400
- Get-ChildItem
 - codeSigningCert 384
 - filter 273
 - include 209, 273
 - name 211, 219
 - Platzhalterzeichen 244
 - recurse 209
- Get-Command 285, 498, 508
 - noun 321
- Get-Content 24–25, 232
 - delimiter 26
 - readCount 233
 - wait 257
- Get-Credential 541
- GetDataSources 448–449
- Get-Date 72, 79
 - format 73
- GetDetailsOf 238
- GetDevices() 320
- Get-EventLog 336
 - asString 336
 - list 336
 - newest 339
- Get-ExecutionPolicy 175, 383
- GetFactoryClasses 430
- GetFolderPath 222
- GetFullPath 215–216
- Get-Item 89, 217, 240, 275
- Get-ItemProperty 274
- Get-Job 536
- Get-LocalGroup 478
- Get-LocalGroupMembers 478
- Get-LocalGroups 481
- Get-LocalUser 476
- Get-LocalUsers 480
- Get-Location 246
- Get-Member 124
- Get-Module 495
- GetName 440
- Get-PfxCertificate 388
- Get-Process 294
- Get-PSDrive 288, 499
- Get-PSProvider 499
 - registry 287
- Get-PSSessionConfiguration 520
- Get-PSSnapin 494
- Get-Service 316, 322
- Get-SoftwareUpdates 502
- GetUnresolvedProviderPathFromPSPath 248–249
- GetValue 275, 278, 440
- Get-Verb 497
- Get-WinEvent 337, 341, 356
- Get-WmiObject 126, 160
 - filter 344
 - Win32_NTEventLogFile 354
 - Win32_NTLogEvent 344
 - Win32_Service 318
- Globaler Katalog 465
- Globally Unique Identifier (GUID) 39
- greedy 56
- grep 142
- Groß- und Kleinschreibung
 - Operatoren 41
 - Reguläre Ausdrücke 50
 - Text umwandeln 58

GroupInfo-Objekt 138
Group-Object 136, 351–352
 noElement 352
groupType 468
Gruppen
 Mitglieder 473–474
gt 149
GUID 39

H

HashMismatch 400
Hashtable 115
 in Objekt verwandeln 441
 sortieren 117
HasMoreData 536
hasPrivateKey 403
Here-String 19, 29, 420, 422
Hilfeinformationen, Skript oder Funktion 190
History 223
HomeDrive 222
HotfixID 124
Hour 79

I

IBM DB2 436
icaccls.exe 364, 368, 374
ICMP-Requests 311
icontains 149
IdentityReference 365, 367
ieq 149
If 153
ige 149
igt 149
ile 149
ilt 149
Import 387
Import-Alias 308
Import-Clixml 114, 332, 418, 423
Import-Csv 422, 460
Import-Module 496
ImportNode 427
Import-PSSession 522
Indexzahlen 101
ine 149
Informix 436
Ingres 436
InheritanceFlags 367
innerXML 428
inotcontains 149

InputBox 23
Insert 111
Insert into 441
InsertionStrings 345
Integrated Scripting Environment 172
Interbase 436
Internet Explorer 300
InternetCache 223
Internet-Cookies 223
Internet-Favoriten 223
InvocationInfo 199
Invoke-Command 514
InvokeGet 472
InvokeMethod 325
InvokeSet 472
IPAddress 126
IP-Adressen
 generieren 130
 prüfen 40
IPEnabled 126
is 147, 149
isAccountLocked 490
ISE 522
iSeries 436
IsInherited 367
isOnline 311
Item 209, 440
ItemProperties 267

J

Jagged Arrays 98
join 59–60, 102
Join-Path 61, 218

K

Kalenderwoche 73
KB-Artikelnummer 124, 146
Keep 536
Kennwort
 in Skript angeben 302
 verschlüsseln 302
 zufällig generieren 59
kommaseparierte Liste (XML) 420
Kommentarblock 190
Kopieren
 Dateien 224
 Ordner 225

L

Last 414
 LastLogonTimestamp 470
 LastWriteTime 130
 Laufwerk finden 49
 lazy 56
 LDAP
 Abfragesprache 458, 466
 Filter 467
 le 149
 Leerzeichen 54
 entfernen 53
 like 40–41, 108
 literalPath 211, 244
 Load (XML) 426, 445
 LoadUserProfile 302
 LoadWithPartialName 387
 LocalAppData 222
 LocalApplicationData 223
 Logische Operatoren 151
 Lotus Notes 434
 ls 208, 308
 lt 149

M

makecert.exe 390–391
 ManagementObject 327
 Mandatory 183
 Mandatory Properties 484
 Manifest 504
 MARS 439
 match 42, 46, 50–51, 56, 109, 146
 md *siehe* New-Item
 MDAC-Komponente 430
 Measure-Command 210
 Measure-Object 105
 property 105
 Member 475
 MemberOf 473
 Memory Leaks 306
 Message 351
 MessageData 543
 Metadaten 237
 lesen 237
 Microsoft Software Development Kit 389
 Microsoft SQL Server 438
 Microsoft.PowerShell 518
 Microsoft.PowerShell.Commands.FileSystemCmdl
 etProviderEncoding 231
 Microsoft.VisualBasic.Interaction 24

Millisecond 79
 Mimer SQL 434
 MinimumRetentionDays 360
 Minute 79
 mod 416
 ModifyOverflowPolicy 359
 Modul
 erneut importieren 497
 Modul-Manifest 504
 Month 79
 Move-Item 224
 MoveNext 446
 MsgBox 24
 msNPAllowDialin 467
 Multiline-Modus
 Regulärer Ausdruck 52
 Multiple Active Result Sets 439
 MultiString 270
 Musik-Ordner 223
 MyComputer 223
 MyDocuments 223
 MyMusic 223
 MyPictures 223
 MySQL 436

N

Nachkommastellen 32–33
 name 415
 Namespace 238
 ne 112, 149
 net.exe 320
 Netzwerkadapter 125
 New-ADOrganizationalUnit 468
 New-ADUser 458, 460
 New-CodeSignCert 391
 New-IntervalJob 538–539
 New-Item 226
 Registrykey 268
 type 227, 268
 type File 228
 New-ItemProperty
 force 280
 New-LocalUser 483
 New-ModuleManifest 504
 New-Object 100, 387, 441, 445
 New-PSDrive 288, 500
 New-PSSession 521
 New-Service 322, 332
 New-TimeSpan 23, 88–89
 NoClobber 63

noElement 138
 noExit 176, 305
 noProfile 176, 305
 Nordwind.mdb 431
 Normalize-Space 415
 not 149, 152, 415
 notcontains 149, 152
 NotifyFilter 256
 notlike 161
 NotSigned 400
 Now 72
 NTLM 525
 ntuser.dat 291

O

Objekteigenschaften
 an Parameter binden 179
 ODBC 430
 Connection Strings 433
 OLE DB 430, 435
 Connection String 438
 Provider 438
 Treiber 435
 Open Database Connectivity 430
 OpenRemoteBaseKey 290
 OpenSubKey 281–282
 Operatoren 41
 logische 150
 Option Constant 195
 Option ReadOnly 196
 or 125, 151
 Oracle 434, 437
 Ordner
 aktueller 246
 Änderungen überwachen 252
 Inhalt auflisten 208
 kopieren 225
 neu anlegen 226
 umbenennen 235
 vertrauenswürdiger 299
 Zugriffsrechte 364
 Organisationseinheit 468
 Out-File 62, 228, 231
 append 231
 encoding 229
 Out-GridView 347
 Out-String 25, 27, 141, 232
 Owner 365

P

Paradox 437
 param()-Block 181
 Parameter
 Aliasname 180
 an Objekteigenschaften binden 179
 mehrere Werte übergeben 187
 optionale 181
 Switch 184
 Typ zuweisen 184
 validieren 186
 von Pipeline empfangen 179
 zwingender 183
 ParseExact() 85
 Parsen
 Ereignisprotokoll 64
 ParseName 238
 passThru 459
 Password 435
 zufällig generieren 59
 Path 176
 Performance-Diagnose 352
 PersistKeySet 388
 Pervasive 437
 Pfadname
 auf Platzhalterzeichen prüfen 217
 auflösen 214, 248
 Bestandteile auswerten 219
 Dateiname extrahieren 219
 der Systemordner 220
 erstellen 61
 in Einzelinformationen zerlegen 50
 konstruieren 218
 Laufwerk 219
 relativ 213
 Pfx 402
 Ping.exe 308, 311
 PKI *siehe* Public Key-Infrastruktur
 Platzhalterzeichen 41
 Dateisystem 245
 Reguläre Ausdrücke 42
 Pop-Location 247, 262
 stackname 247
 Position 416
 powershell.exe 176, 305
 preceding-sibling 416
 Privater Schlüssel 402
 Privater Schlüssel, exportierbar 386
 Process-Block 133, 177–178, 197
 ProcessID 327
 Profilkript 177, 308

ProgramData 222
 ProgramFiles 222–223, 300
 Programme-Menü 223
 Programme-Ordner 300
 Programs 223
 PropagationFlags 367
 Provider 435
 Provider-Factory 439
 Prozess
 abgestürzte finden 307
 abgestürzter 307
 als anderer Benutzer ausführen 301
 Anzahl Instanzen 298
 auflisten 294
 beenden 305
 feststellen, ob läuft 297
 mit vollen Administratorrechten 304
 nach Firma 297
 nach Laufzeit 296
 starten 299
 PSBase 325, 412
 PSBase.CommitChanges 485
 PSIsContainer 197
 PSObject 290
 PSParentPath 404
 Public 222
 Public Key-Infrastruktur (PKI) 390
 Push-Location 247, 262
 stackname 247
 Put 472, 484
 PutEx 472
 Pwd 433

Q

Quantifizierer 43
 Reguläre Ausdrücke 43
 QWord 270

R

Read 440
 ReadCount 25, 233
 Read-Host 22
 ReadOnly 243
 Receive-Job 533, 536
 Recent 223
 recurse 212
 reg.exe 291
 RegEx 48, 50
 Register-ObjectEvent 539, 543

Register-PSSessionConfiguration 519, 527
 Registrierungsdatenbank
 Byte-Array 59
 Registrierungsdatenbank (Remotezugriff) 289
 Registrierungs-Editor öffnen 285
 Registrierungsschlüssel *siehe* Schlüssel
 Registry *siehe* Registrierungsdatenbank
 RegistryRights 367, 376
 Reguläre Ausdrücke 42
 Dateien 212
 greedy, lazy 56
 Platzhalterzeichen 42
 Quantifizierer 43
 Sonderzeichen entwerfen 48
 Textanker 44
 Relativer Pfadname 213
 RemoteSigned 175, 383
 Remove-ADGroupMember 475
 Remove-ADUser 462
 Remove-Cert 404
 Remove-IntervalJob 538
 Remove-Item 194, 234, 273
 confirm 273
 force 235
 recurse 273
 Sicherheitsabfrage 273
 Zertifikatspeicher 404
 Remove-ItemProperty 283–284
 Remove-Job 533, 537
 Remove-LocalGroup 486
 Remove-LocalUser 491
 Remove-PSSession 521
 Rename-Item 236
 newname 236
 replace 236
 replace 51, 54–55, 471
 ReplacementStrings 344
 reset 463
 ResetAccessRule() 375
 Resolve-Path 215
 Responding 307
 Restart-Service 322, 326
 Restricted 175, 384
 Resume-Service 322
 Robocopy 224, 226
 Root-Zertifikat 389
 RSS 410
 Rückgabewert
 Skript 180
 Rückverweise 52, 55
 RunAs 304
 Run-NativeApp 311

S

- SAM-Datenbank 477
- Save (XML) 426
- Schleife 160–161
 - Arrayelemente 103
- Schlüssel
 - anlegen, wenn nicht vorhanden 271
 - exportierbarer 386
 - lesen 274
 - löschen 273
 - mit Schreibrechten öffnen 281
 - neuen anlegen 268
 - privater 403
 - prüfen, ob vorhanden 279
 - Standardwert 276
 - Standardwert lesen 278
 - Standardwert löschen 284
 - suchen 272
 - Unterschlüssel auflisten 265
 - Wert ändern 280
 - Wert hinzufügen 280
 - Wert löschen 282
 - Zugriffsrechte 364
- Schlüssel-Wert-Paare 115
- Schreibschutz-Attribut 240
- Scope 175
- SDDL-Form 377
- SDDL-Syntax 374
- SDK (Software Development Kit) 389
- SearchBase 465
- SearchScope 465
- Second 79
- SecureString 461, 463
- Security Descriptor 364, 374
- Security Descriptor Definition Language 374
- Security Descriptor *siehe* Sicherheits-
beschreibungen
- SelectNodes 412–413
 - Xpath 445
- Select-Object 128, 330, 469
- Select-String 232
- Senden an-Menü 223
- SendTo 223
- Server 433, 435
- ServiceController 318
- ServicesDependedOn 323
- SetAccessRule() 375
- SetAccessRuleProtection() 376
- Set-Acl 364, 368, 374, 376–377
- Set-ADAccountPassword 463
- Set-ADUser 471
- Set-Alias 307–308
- Set-AuthenticodeSignature 397–398
- Set-Content 230–231
 - encoding 229–230
- Set-ExecutionPolicy 175
- SetInfo 484
- Set-Item 195
- Set-ItemProperty 280
- Set-Location 246, 273, 355
- SetPassword 463, 484
- Set-PSSessionConfiguration 518, 520
- SetSecurityDescriptorSddlForm 368, 377
- Set-Service 322, 324
- SetValue 282
- Set-Variable 543
- Shell.Application 237–238
- Show-EventLog 347
- Sicherheitsabfrage abschalten 462
- Sicherheitsbeschreibungen 365
- Sicherheitseinstellung per Skript 174
- SideIndicator 113, 251
- SignerCertificate 399
- SilentlyContinue 211, 273, 298
- Skript 172
 - automatisch ausführen 176
 - extern starten 174
 - Hilfe hinzufügen 190
 - mit Administratorrechten starten 304
 - remote ausführen 514
 - Signatur prüfen 399
 - signieren 396
- Skriptblock 192, 236
- SmartCards 390
- Software Development Kit (SDK) 389
- Sonderzeichen 18–19, 29
- Sort-Object 106, 133, 237, 352
 - descending 106
 - Hashtable 118
 - IP-Adressen 534
 - Versionen 534
- Spaltenüberschriften
 - abweichende 128
- Split 46, 48
- Split-Path 50
 - isAbsolute 217
 - parent 219
 - resolve 219
- SQL Express 439
- SQL Server 434, 437, 448
- SQLBase 438
- SqlRecord 441

- Stammzertifizierungsstellen
 - vertrauenswürdige 391, 395
- Standardordner 355
- Standardsicherheit
 - Datenbank 439
- Start-Bitstransfer 498
- StartMenu 223
- Startmenü 223
- Start-MonitoredJob 541
- Start-Process 301
- Start-Service 322
- StartsWith 41, 415
- StartTime 296, 340
- Startup 223
- Startvorgänge 347
- StateChanged 543
- Stichwort in Text enthalten 146
- Stop-Computer 132
- Stop-Job 537
- Stop-Process 305–306
- Stop-Service 322
- StreamReader 259
- String *siehe* Text
- StringBuilder 62
- String-Length 413, 416
- SubString 45, 47, 282
- Suspend-Service 322
- Switch 64, 66, 153, 167, 184
 - Parameter 184
 - regex 67
- Sybase 438
- SynchronizingObject 254
- System 223
- System.Collection.ArrayList 110
- System.Data.Odbc 433
- System.Data.OleDb 435
- System.Diagnostics.EventLog 338
- System.Guid 39
- System.IO.DirectoryInfo 197
- System.IO.File 27, 164
- System.IO.Path 220
- System.Management.ManagementObject 90
- System.Text.StringBuilder 62
- System.Version 534
- Systemordner 223
 - Pfad bestimmen 222
 - Pfadnamen ermitteln 220
- Systemticks 91, 470

T

- Tabulatorzeichen 46
- TakeOwnership() 378
- TargetObject 200
- Tee-Object 533
- telephoneNumber 472
- Temp 222
- Templates 223
- Teradata 438
- Test-Connection 130, 312, 533
- Test-LocalAccount 482
- Test-Path 271
- Testzertifikate
 - erstellen 391
- Text 427
 - aus Datei lesen 24
 - doppelte Wörter entfernen 55
 - erfragen 22
 - ersetzen 51
 - Ersetzungen vornehmen 51
 - formatieren 33
 - in Sätze aufgliedern 49
 - in Variable speichern 18
 - in Zeichen verwandeln 58
 - Informationen einfügen 29
 - Länge bestimmen 41
 - Leerzeichen entfernen 53
 - splitten 46
 - Stichwort finden 39
 - Teile daraus lesen 45
 - wiederholen 32
 - Wortbereiche finden 56
- Textanker 51
 - Reguläre Ausdrücke 44
- Textbereich
 - finden 56
- Textdatei 62
 - abgeschnittene Informationen 63
 - Breite angeben 63
 - parsen 64
- Text-Encoding 230
- Textfeld 24
- Textlisten
 - umformatieren 61
- Textmuster 41–42
- Textschablone 34
- Throw 203
- Thumbprint *siehe* Fingerabdruck
- Ticks 79, 91
- TimeGenerated 128
- TimeOfDay 79

Timespan 328
 Tmp 222
 toLower 41, 58, 418
 toString 38, 62
 toUpper 58
 Trap 200
 Trim 53
 TrimEnd 53
 TrimStart 53
 Trusted Connection 439
 try-Block 198
 TryCast 86
 Type Accelerator 90, 327

U

UFormat 73
 Uid 433
 Umbruch
 Code 173
 Umgebungsvariablen 176, 221
 Inhalt verwenden 222
 Path 300
 Unicode-Format 63, 230
 UnknownError 400
 Unregister-Event 539
 Unrestricted 175, 383
 Unterausdruck 31, 97, 422
 Upates, installierte anzeigen 343
 Update 448
 User ID 435
 UserProfile 222
 UTF8 63

V

Valid 400
 ValidateCount 188
 ValidatePattern 187
 ValidateRange 186
 ValidateScript 187
 ValidateSet 186
 Validierungsattribute 186
 ValueFromPipeline 179
 ValueFromPipelineByPropertyName 179
 Variablen auflösen 18
 Verb 304
 Vererbung 376
 Einstellungen 376
 Vergleichsoperatoren 146
 Verify 393

Verlaufsdaten im Browser 223
 Verteilergruppen 468
 Vertrauenswürdige Ordner 299
 Vervollständigungsmodus 20
 Verwendungszweck
 Zertifikat 385
 VisualBasic 24
 Void 110
 Vorher-Nachher-Zustand 114
 Vorlagen 223

W

WaitForChanged 252
 WaitForStatus 326
 Wait-Job 533, 541
 whatIf 235, 283, 306
 Where-Object 123, 153, 211, 242, 417
 While 161–162
 Widening 148
 Width 63
 Win32_Directory 378
 Win32_Service 322
 Delete 332
 Win32_UserAccount 160
 Windir 222
 Windows Backup 349
 Windows PowerShell ISE 173
 Windows-Performance-Diagnose 352
 Windows-Start 347
 windowsupdate.log 26
 WMI
 Datum umwandeln 345
 Datum und Zeit umwandeln 90
 Ereignisprotokolle 344
 Wortgrenze 55
 WQL 126
 Wrap 129

X

X509Certificates.StoreName 395
 X509Certificates.X509Certificate2Collection 387
 X509Certificates.X509ContentType 401
 X509Certificates.X509KeyStorageFlags 387
 X509Certificates.x509Store 387
 X509Chain-Objekt 393
 X509KeyStorageFlag 387
 xcopy 226
 XML 408, 422
 Schablone 420

xor 152
XPath 412–413, 445

Y

Year 79

Z

Zahlenbereiche 97
Zahlenformatierung 38
Zeilenumbruch 21, 28, 173
 Text ohne ~ anfügen 63
Zeitdifferenz 22
Zeitstempel 34, 79

Zeitunterschied 87
Zertifikat 385
 erstellen 389, 391
 exportieren 401
 für vertrauenswürdig erklären 394
 importieren 386
 Importproblem in Vista 395
 löschen 403
 privaten Schlüssel exportieren 402
 selbstsigniert 394
 überprüfen 393
Zugriffsrechte
 Datei 364
 Registrierungsdatenbank 364
Zuletzt verwendeten Programme 223

Der Autor

Dr. Tobias Weltner ist einer der bekanntesten Skriptexperten und Mitbegründer der PowerShell-Communities www.powershellcommunity.org und www.powershell.com. Von ihm stammen die Skript-Entwicklungsumgebungen *SystemScripter* (VBScript) und *PowerShellPlus*. In mittlerweile mehr als 100 Fachbüchern zeigt er seine Leidenschaft, technisch anspruchsvolle Themen klar, einfach und mit hohem Praxiswert zu erklären. Als deutscher *Microsoft MVP für PowerShell* hält er den engen Kontakt zur PowerShell-Produktgruppe in Redmond und spricht auf Fachkonferenzen und Roadshows. Dr. Weltner führt regelmäßig europaweit Workshops und Trainings zu Windows PowerShell durch und bildet Mitarbeiter und Trainer mittlerer und großer Unternehmen aus. Wollen auch Sie einen Inhouse-Workshop mit ihm organisieren, erreichen Sie ihn unter tobias.weltner@email.de.