

X-systems.press

Peter Monadjemi

# Windows Server- Administration mit PowerShell 5.1

Eine kompakte und  
praxisorientierte Einführung

**EBOOK INSIDE**

 Springer Vieweg

---

**X.systems.press**

**X.systems.press** ist eine praxisorientierte Reihe zur Entwicklung und Administration von Betriebssystemen, Netzwerken und Datenbanken.

Weitere Bände in dieser Reihe  
<http://www.springer.com/series/5189>

---

Peter Monadjemi

# Windows Server-Administration mit PowerShell 5.1

Eine kompakte und praxisorientierte  
Einführung

Peter Monadjemi  
Esslingen, Deutschland

ISSN 1611-8618

X.systems.press

ISBN 978-3-658-17665-5

DOI 10.1007/978-3-658-17666-2

ISSN 2363-9059 (electronic)

ISBN 978-3-658-17666-2 (eBook)

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Springer Vieweg

© Springer Fachmedien Wiesbaden GmbH 2017

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature

Die eingetragene Gesellschaft ist Springer Fachmedien Wiesbaden GmbH

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

---

## Vorwort

Der Satz „Veränderungen machen auch vor den IT-Abteilungen nicht Halt“ mag zunächst widersinnig erscheinen, steht doch die IT-Branche selber wie kaum eine andere Branche für die Veränderung schlechthin. Doch bei etwas genauerer Betrachtung ergibt sich ein differenzierteres Bild. In vielen großen Organisationen dominieren statische Prozesse, eine strikte Aufgaben- und Kompetenzaufteilung und damit verbunden auch Formalismen, Bürokratie und eine gewisse „Das haben wir immer schon so gemacht“-Mentalität. Es versteht sich von selber, dass diese Strukturen für die modernen Herausforderungen im Cloud-Zeitalter nicht optimal sind. Eine dieser Herausforderungen besteht darin, dass die in den letzten zehn Jahren und länger im Bereich der Software-Entwicklung entstandene Agilität nicht durch starre Strukturen in der IT-Abteilung ausgebremst werden soll.

Das exorbitante Wachstum der Internet-Giganten, allen voran Amazon, Facebook, Twitter, aber auch viele nicht ganz so in der Öffentlichkeit bekannte Größen wie Flickr, Booking.com, Ueber und viele, viele mehr sind nicht nur in kurzer Zeit gewachsen, sondern haben zwangsläufig ihre eigenen IT-Prozesse definiert. In diesen Prozessen fließen Entwicklung und Bereitstellung zwangsläufig zusammen und erfordern ein Umdenken. Auch Microsoft hat einen enormen Aufwand, verbunden mit einer tiefgreifenden internen Umstrukturierung, betrieben, um den Anschluss nicht zu verlieren, und die Ergebnisse in Gestalt neuer Cloud-Produkte, allen voran Office 365, und der Aktienkurs scheinen zu bestätigen, dass man auf dem richtigen Weg ist und die Dominanz auf dem Desktop in der Cloud behaupten kann.

Um das Zusammenwachsen von „Development“ und „Operations“ zu beschreiben, wurde bereits vor vielen Jahren der Begriff „DevOps“ geprägt. Anders als man es vermuten könnte, entstand dieser Begriff weder in der Marketingabteilung eines Internet-Giganten, noch stammt er aus der Doktorarbeit eines Informatikers. Der Begriff entstand, ganz unspektakulär, eher zufällig, als eine belgische Usergroup einen griffigen Namen für eine geplante Veranstaltungsserie suchte.

Auch wenn ich den Begriff „DevOps“ nicht überbewerten möchte, steht das Kürzel doch stellvertretend für den grundlegenden Wandel, der in der IT-Branche seit einigen Jahren im vollen Gange ist, und der noch längst nicht abgeschlossen ist (dazu ist die weltweite IT-Landschaft auch zu groß). Es ist auch nicht so, dass DevOps für alle IT-Abteilungen

einen nicht zu vermeidenden Paradigmenwechsel bedeutet. Es hängt sehr stark davon ab, ob zum Beispiel das Thema Bereitstellen von (Web-)Anwendungen überhaupt ein Thema ist. Der kleinste gemeinsame Nenner von DevOps steht einfach für mehr Agilität, schlankere Prozesse (das klingt natürlich immer gut) und damit verbunden für die Aufgabe starrer Strukturen.

Alles schön und gut, doch welche Rolle spielt die PowerShell? Die PowerShell ist natürlich kein „DevOps“-Tool, das es genauso wenig gibt wie ein „TCO-Tool“, mit dem sich die „Total Cost of Ownership“ reduzieren lässt.

Die mit der Version 4.0 eingeführte *Desired State Configuration* (DSC) ist eine Technik, die einen zentralen Aspekt von DevOps adressiert: Das schnelle Bereitstellen von Konfigurationsänderungen. Das moderne Schlagwort lautet „Infrastruktur als Code“, also die textuelle Beschreibung von herzustellenden Zuständen in einer modernen IT-Infrastruktur. Insbesondere im Zusammenspiel mit den Cloud-Plattformen von Microsoft aber auch Amazon lassen sich agile Prozesse wie das Bereitstellen von Anwendungen und Konfigurationsänderungen flexibel anwenden. DSC wurde mit den Versionen 5.0 und 5.1 weiterentwickelt und steht auch im Rahmen der Cloud-Plattform Azure zur Verfügung. DSC ist „work in progress“. Microsoft plant für die kommenden Versionen wichtige Verbesserungen beim *Local Configuration Manager* (LCM), dem Dreh- und Angelpunkt von DSC, die zu dem Zeitpunkt als dieses Buch entstand, aber noch nicht angekündigt waren.

---

## Open Source und PowerShell „on every System“

Am 16. August 2016 wurde vom PowerShell-Team die vielleicht wichtigste Ankündigung in der inzwischen zehn Jahre dauernden Geschichte der PowerShell gemacht. An diesem Tag wurde aus der PowerShell ein Open Source-Projekt. Zwar bezieht sich diese Ankündigung „nur“ auf die PowerShell Core Edition, die in der Version 6.0 zu diesem Zeitpunkt lediglich in einer frühen Alpha-Version vorliegt und voraussichtlich irgendwann in naher Zukunft offiziell werden wird, dennoch markiert diese Ankündigung einen wichtigen Richtungswechsel. In naher Zukunft wird es die PowerShell in der Core Edition nicht nur für Windows, Mac OS X und die wichtigsten Linux-Distributionen geben, sondern für jede Plattform, auf der sich die Laufzeitumgebung .NET Core implementieren lässt.

---

## Was lesen Sie in diesem Buch?

Eine „Warnung“ gleich vorweg: Dieses Buch ist kein typisches PowerShell-Einsteigerbuch. Falls Sie als Administrator oder IT-Pro die PowerShell zunächst kennenlernen möchten, ist dieses Buch vermutlich keine gute Wahl, da es bereits Grundkenntnisse im Umgang mit der PowerShell voraussetzt. Begriffe wie Cmdlet, Parameter, Pipeline, Functions und vor

allem Objekte sollten beim Lesen einen vertrauten Klang besitzen und keine virtuellen Fragezeichen auslösen. Ich möchte mit diesem Buch eine Lücke schließen zwischen der umfangreichen Einstiegsliteratur, in der die Grundlagen der PowerShell leicht verständlich erklärt werden, und den Blog-Einträgen der PowerShell-Experten, in denen in der Regel Techniken verwendet werden, die scheinbar „nirgendwo“ erklärt werden (viele findet man in der umfangreichen PowerShell-Hilfe und den weit über 100-About-Themen, doch seien wir einmal ehrlich, wer liest gerne die Hilfe?). Auf insgesamt 19 Kapiteln dieses Buches erhalten Sie einen Querschnitt über die wichtigsten Themen, die in die Kategorie Grundlagen und Praxis für den etwas fortgeschrittenen Einsatz der PowerShell fallen. Dass das Buch mit dem Kapitel „Spaß mit der PowerShell“ endet, geschah nicht ohne Grund. Die PowerShell ist nicht nur ein Werkzeug für die tägliche Arbeit, sondern auch eine universelle Skriptsprache in der Tradition von Perl, Python, Bash usw., mit der sich einfach eine Menge machen lässt. Und wer möchte, kann mit ihr auch Spaß haben.

---

## Die Beispiele zu diesem Kapitel

Einige Beispiele, die ich in diesem Buch vorstelle, sind aus Platzgründen nicht vollständig abgebildet. Sie erhalten alle Beispiele als Download. Die Adresse finden Sie in meinem Blog <http://poshadmin.de>. Sollten Sie dort wider Erwarten nicht fündig werden, schicken Sie mir bitte eine E-Mail und schreiben Sie einen Eintrag in das Forum.

---

## Kontakt zum Autor

Ich freue mich über Lob, Kritik und Fragen zum Buch oder allgemein zur PowerShell. Sie erreichen mich per E-Mail unter [pm@activetraining.de](mailto:pm@activetraining.de), per Twitter (@pemo09) oder über die Kommentarfunktion meines Blog <http://poshadmin.de>.

---

## Danksagungen

Ich möchte mich bei meiner Lektorin, Frau Dr. Kathke vom Springer-Vieweg-Verlag, für die wirklich gute Zusammenarbeit bedanken (ursprünglich hätte das Buch natürlich deutlich eher erscheinen sollen). Die persönliche Widmung ist für meine Frau Andrea in tiefer Dankbarkeit für die vergangenen 13 Jahre. Ohne ihre Unterstützung und ihre unglaubliche Fähigkeit, in jedem Chaos eine Struktur zu erkennen und in verfahrenen Situationen einen klaren Kopf zu bewahren, wäre aus diesem Buch nichts geworden.

Peter Monadjemi  
Esslingen, Mai 2017



---

# Inhaltsverzeichnis

<b>1 PowerShell für Kurzschnellgeschlossene</b> .....	1
1.1 Das WMF im Überblick .....	1
1.2 Die .NET-Laufzeit .....	3
1.2.1 Assemblys .....	3
1.3 Die Objekt-Pipeline .....	5
1.4 PSProvider und PSDrives .....	6
1.4.1 Dynamische Parameter .....	7
1.5 Functions, Aliase, Workflows und Configurations .....	8
1.6 Erweiterbarkeit .....	8
1.6.1 Ein Modulmanager in Gestalt des PowerShellGet-Moduls .....	10
1.7 Befehlssyntax .....	11
1.8 Die moderne Konsole .....	12
1.9 Hilfe .....	13
1.10 Andere Plattformen .....	14
<b>2 Der Umgang mit Typen</b> .....	15
2.1 Alles ist ein Objekt .....	15
2.1.1 Objekte basieren auf Typen .....	15
2.1.2 Den Typ eines Objekts herausfinden .....	16
2.2 Typen als Objekte .....	16
2.3 Die Typen eines Arrays erhalten .....	17
2.3.1 Auflisten der Konstruktoren einer Klasse .....	17
2.4 Neue Objekte anlegen mit dem New-Object-Cmdlet .....	19
2.4.1 Neue Objekte anlegen über die statische Methode New .....	19
2.4.2 Objekte mit Argumenten anlegen .....	19
2.4.3 Neue Objekte anlegen per [PSCustomObject] .....	20
2.5 Objektschreibweise .....	20
2.6 Das versteckte Member PSObject .....	21
2.7 Typen erweitern .....	21

2.8	Objekte erweitern über das Add-Member-Cmdlet . . . . .	23
2.9	Typen definieren über externen Code . . . . .	24
2.9.1	Zusammenfassung . . . . .	26
<b>3</b>	<b>Klassen definieren mit dem <i>class</i>-Befehl</b> . . . . .	<b>27</b>
3.1	Einleitung . . . . .	27
3.1.1	Befehlswörter für die Definition von Klassen . . . . .	28
3.2	Klassen definieren . . . . .	28
3.3	Hinzufügen von Eigenschaften . . . . .	29
3.4	Aus Klassen Objekte machen . . . . .	31
3.5	Statische Member . . . . .	31
3.6	Enumerationen . . . . .	32
3.7	Aus Klassen werden Objekte . . . . .	33
3.8	Klassen mit einem Konstruktor . . . . .	34
3.8.1	Konstruktor mit Parameter . . . . .	35
3.8.2	Versteckte Members . . . . .	35
3.8.3	Überladene Konstruktoren . . . . .	37
3.8.4	Klassen mit einem statischen Konstruktor . . . . .	38
3.9	Methoden definieren . . . . .	38
3.9.1	Methoden überladen . . . . .	40
3.9.2	Statische Methoden . . . . .	41
3.10	Klassen ableiten (Vererbung) . . . . .	42
3.10.1	Ableiten mit einem Konstruktor, der nicht parameterlos ist . . . . .	44
3.10.2	Ableiten von Klassen der .NET-Klassenbibliothek und der PowerShell-Bibliotheken . . . . .	45
3.10.3	Members überschreiben . . . . .	45
3.11	Typenformatierung mit eigenen Klassen . . . . .	49
3.12	Zusammenfassung . . . . .	51
<b>4</b>	<b>Functions für Fortgeschrittene</b> . . . . .	<b>53</b>
4.1	Was macht eine Funktion „Advanced“? . . . . .	53
4.1.1	Die Rolle der Attribute . . . . .	53
4.1.2	Das Prinzip der Parameter-Zuordnung . . . . .	54
4.2	Das CmdletBinding-Attribut . . . . .	55
4.3	Das Parameter-Attribut . . . . .	56
4.4	Die Parameterbindung im Detail . . . . .	57
4.4.1	Wenn die Parameterbindung nicht funktioniert . . . . .	58
4.4.2	Die Parameterbindung sichtbar machen . . . . .	59
4.5	Functions mit Pipeline-Parametern . . . . .	59
4.6	Functions, die die Pipeline abarbeiten . . . . .	61
4.7	Functions mit einer eingebauten Bestätigungsanforderung . . . . .	62
4.7.1	Die Auswirkung des Confirm-Parameters . . . . .	63
4.7.2	Die Auswirkung des WhatIf-Parameters . . . . .	63

---

4.7.3	Implementieren von Confirm und WhatIf an einem Beispiel . . . . .	63
4.7.4	Eine Bestätigungsanforderung unabhängig von Confirm & Co . . . . .	66
4.8	Parameter-Validierung . . . . .	67
4.8.1	Einen Bereich validieren mit ValidateRange . . . . .	67
4.8.2	Eine Auswahlmenge validieren mit ValidateSet . . . . .	67
4.8.3	Ein beliebiges Kriterium validieren per ValidateScript . . . . .	67
4.8.4	Credential-Parameter implementieren . . . . .	68
4.8.5	Eigene Parameterattribute definieren . . . . .	68
4.9	Dynamische Parameter . . . . .	69
4.10	Zusammenfassung . . . . .	71
<b>5</b>	<b>Aus Functions und Skripte werden Module . . . . .</b>	<b>73</b>
5.1	Module und Modultypen . . . . .	73
5.1.1	Skriptmodule . . . . .	74
5.1.2	Manifestmodule . . . . .	74
5.1.3	Binärmodule . . . . .	74
5.1.4	Dynamische Module . . . . .	75
5.1.5	Weitere Modulverzeichnisse hinzufügen . . . . .	76
5.2	Manifestmodule im Detail . . . . .	76
5.3	Verschachtelte Module . . . . .	78
5.4	Mehrere Versionen eines Moduls verwenden . . . . .	78
5.5	Internationale Module – Zeichenketten in Psd1-Dateien auslagern . . . . .	79
5.5.1	Die Rolle der Kultur und das CultureInfo-Objekt . . . . .	81
5.5.2	Ändern der aktuellen Kultur . . . . .	82
5.6	Praxisteil: Erstellen eines Manifestmoduls . . . . .	83
5.7	Zusammenfassung . . . . .	85
<b>6</b>	<b>PowerShell-Skripte testen mit Pester . . . . .</b>	<b>87</b>
6.1	Was testet Pester? . . . . .	87
6.1.1	Testen und DevOps . . . . .	88
6.2	Das Pester-Modul im Überblick . . . . .	88
6.3	Die ersten Schritte mit Pester . . . . .	89
6.4	Einen Test-Rahmen mit New-Fixture anlegen . . . . .	92
6.5	Test-Ergebnisse vergleichen mit Should – die Rolle der Assertions . . . . .	93
6.6	PowerShell-Commands nachbilden über Mocks . . . . .	95
6.6.1	Feststellen, ob ein Mock-Command ausgeführt wurde . . . . .	97
6.7	Tests in einen Ablauf einbeziehen . . . . .	97
6.8	Testen als (Lebens-)Philosophie . . . . .	97
6.9	Zusammenfassung . . . . .	98
<b>7</b>	<b>Skripte und Module bereitstellen . . . . .</b>	<b>99</b>
7.1	Die PowerShell-Paketverwaltung im Überblick . . . . .	99
7.1.1	Ein Blick hinter die Kulissen . . . . .	100
7.1.2	Überblick über das PackageManagement-Modul . . . . .	103

7.1.3	Die ersten Schritte mit der Paketverwaltung . . . . .	103
7.1.4	Anwendungspakete über Chocolatey installieren . . . . .	103
7.1.5	Package-Provider offline installieren . . . . .	107
7.1.6	Das PowerShellGet-Modul für die Modul- und Skriptverwaltung . . . . .	107
7.1.7	Die ersten Schritte mit PowerShellGet . . . . .	107
7.2	Eigene Ablagen für Module und Skripte einrichten . . . . .	109
7.2.1	Skripte über GitHub als „Gists“ abrufen . . . . .	112
7.2.2	Repository statt Webverzeichnis . . . . .	112
7.2.3	Einrichten eines Modul-Repositorys mit MyGet. . . . .	113
7.2.4	Skripte und Module in der PowerShell Gallery veröffentlichen . . .	115
7.3	Arbeiten mit einer Versionsverwaltung . . . . .	117
7.3.1	Schritt für Schritt . . . . .	118
7.3.2	Git-Integration in Visual Studio Code . . . . .	126
7.4	Aufsetzen einer Release-Pipeline für Module . . . . .	129
7.4.1	Was genau ist eine Release-Pipeline? . . . . .	129
7.4.2	Wo gibt es die Release-Pipeline? . . . . .	130
7.4.3	Eine Release-Pipeline selber gebaut . . . . .	131
7.4.4	Eine Release-Pipeline mit AppVeyor. . . . .	134
7.4.5	Die ersten Schritte mit AppVeyor . . . . .	135
7.4.6	Die Anatomie der Yaml-Datei . . . . .	136
7.4.7	Ein PowerShell-Modul per AppVeyor bereitstellen. . . . .	137
7.5	Zusammenfassung . . . . .	139
<b>8</b>	<b>DSC-Grundlagen. . . . .</b>	<b>141</b>
8.1	Ein erstes Beispiel . . . . .	141
8.2	Ein wenig Theorie . . . . .	148
8.2.1	MOF. . . . .	148
8.2.2	DSC-Spracherweiterungen . . . . .	149
8.2.3	Die Rolle der Ressourcen. . . . .	149
8.2.4	Der Local Configuration Manager (LCM). . . . .	150
8.2.5	Pull statt Push. . . . .	151
8.3	Verwenden von Konfigurationsdaten . . . . .	152
8.4	Warum DSC? . . . . .	154
8.5	Zusammenfassung . . . . .	154
<b>9</b>	<b>DSC in der Praxis . . . . .</b>	<b>157</b>
9.1	Auspacken von Zip-Dateien. . . . .	157
9.2	Umgebungsvariablen anlegen . . . . .	158
9.3	Dateien und Verzeichnisse anlegen . . . . .	160
9.4	Lokale Benutzer und Gruppen anlegen . . . . .	161
9.5	Registry-Schlüssel anlegen . . . . .	162

9.6	Windows-Feature installieren	164
9.7	Prozesse starten	164
9.8	Systemdienste einrichten	165
9.9	DSC-Log-Meldungen schreiben	166
9.10	Beliebige Befehle ausführen	168
9.11	Einen Webserver einrichten	169
9.12	Eine Hyper-VM einrichten	173
9.13	Zusammenfassung	174
<b>10</b>	<b>DSC für (etwas) Fortgeschrittene</b>	<b>175</b>
10.1	Hinzufügen von DSC-Ressourcen	175
10.2	Ressourcenabhängigkeiten festlegen	176
10.3	Den LCM konfigurieren	177
10.4	Umgang mit Konfigurationsdaten	178
10.4.1	Konfigurationsdaten unterschiedliche Nodes zuordnen	179
10.4.2	Einzelne Nodes auswählen	180
10.4.3	Allgemeine Eigenschaften in den Konfigurationsdaten festlegen	181
10.4.4	Konfigurationsdaten, die nur allgemeine Einstellungen enthalten	182
10.4.5	Konfigurationsdaten mit strukturierten Werten	183
10.5	Kennwörter in einer Konfiguration verwenden	184
10.6	Einrichten eines Pull Servers	189
10.6.1	Überblick über das Einrichten eines Pull Servers	190
10.6.2	Einrichten eines webbasierten Pull Servers	190
10.6.3	Umstellen des LCM auf den Pull-Modus	195
10.6.4	Einen Pull Server testen	196
10.6.5	Die Rolle der Reportserver	197
10.7	DSC-Diagnose	197
10.8	Eigene Ressourcen definieren	199
10.8.1	Zusammengesetzte Ressourcen (Composite Resources)	203
10.9	Die PowerShell DSC-Cmdlets im Überblick	206
10.10	Zusammenfassung	208
<b>11</b>	<b>Aus Text Objekte machen</b>	<b>209</b>
11.1	Texte im CSV-Format konvertieren	210
11.2	Überschriften nachträglich hinzufügen oder vorhandene Überschriften ändern	211
11.3	Unregelmäßige Texte zerlegen	211
11.3.1	Texte mit dem Split-Operator zerlegen	211
11.3.2	Texte mit Hilfe regulärer Ausdrücke zerlegen	213
11.4	Objekte anlegen	215
11.4.1	Objekte mit dem New-Object-Cmdlet anlegen	215
11.4.2	Objekte mit einer Hashtable anlegen	216

11.4.3	Unregelmäßige Textdaten mit dem ConvertFrom-String-Cmdlet verarbeiten	216
11.4.4	XML-Daten verarbeiten	217
11.5	JSON-Daten verarbeiten	220
11.6	Zusammenfassung	221
<b>12</b>	<b>Active Directory-Administration</b>	<b>223</b>
12.1	AD DS und Domänencontroller einrichten	224
12.1.1	Aktualisieren der Hilfe	225
12.1.2	Authentifizierung	225
12.2	Suche nach Benutzerkonten per Get-AdUser	226
12.3	Die PowerShell-Abfragesyntax	226
12.3.1	Benutzerkonten auswählen über den Identity-Parameter	226
12.3.2	Die Rolle des Properties-Parameter bei Get-ADUser	227
12.3.3	Spezialfall zusammengesetzte Attribute	228
12.3.4	Eingrenzen der Suche	228
12.3.5	Suchen nach anderen AD-Objekten	228
12.4	Benutzer anlegen per New-AdUser	229
12.4.1	Benutzerkonten aktivieren	230
12.4.2	Benutzerkonten über eine CSV-Datei anlegen	230
12.5	Benutzerkonten ändern per Set-ADUser	230
12.5.1	Umgang mit Mehrwert-Attributen	231
12.6	Benutzerkonten löschen mit Remove-ADUser	231
12.7	Nach Kontenattributen suchen	232
12.8	Gruppenzugehörigkeiten verwalten	232
12.8.1	Nicht alles passt zusammen	233
12.9	Computerkonten abfragen per Get-AdComputer	234
12.10	Abfrageergebnisse mit Out-GridView kombinieren	234
12.11	Umgang mit Organisationseinheiten	235
12.12	Das AD-Laufwerk	235
12.13	Den AD-Papierkorb aktivieren	236
12.14	Einen AD LSD oder Open LDAP-Server ansprechen	237
12.15	Zusammenfassung	238
<b>13</b>	<b>Azure-Administration per PowerShell</b>	<b>239</b>
13.1	Ein erster Überblick	239
13.2	Der ARM im Überblick	240
13.2.1	Die Rolle der Resource Provider	240
13.2.2	Die Rolle der Templates	241
13.3	Zugriffssteuerung per BPAC	248
13.4	Die Azure PowerShell im Überblick	249
13.4.1	Abfragen der PowerShell-Version	249
13.4.2	Ein erster Überblick	250

---

13.4.3	Die ersten Schritte mit der Azure PowerShell .....	250
13.4.4	Beispiele aus der Praxis .....	252
13.4.5	Eine virtuelle Maschine über ein Template anlegen .....	257
13.4.6	Azure und DSC .....	258
13.5	Custom Script Extension als Alternative zu DSC .....	262
13.6	Zusammenfassung .....	265
<b>14</b>	<b>Debugging für etwas Fortgeschrittene .....</b>	<b>267</b>
14.1	Unsichtbare Variablen beim Debuggen .....	267
14.2	Der Debug-Modus im Überblick .....	268
14.3	Über das Wesen eines Haltepunktes .....	268
14.3.1	Allgemeine Haltepunkte setzen .....	268
14.3.2	Haltepunkt entfernen und deaktivieren .....	269
14.3.3	Haltepunkte für einen Befehl setzen .....	269
14.3.4	Haltepunkte für eine Variable setzen .....	269
14.3.5	Haltepunkte mit einer Bedingung verknüpfen .....	270
14.3.6	Ein Skript ohne Haltepunkte debuggen .....	270
14.4	Remote-Debugging von Skripten .....	270
14.4.1	Haltepunkte über Invoke-Command setzen .....	271
14.5	Einen Workflow debuggen .....	271
14.6	Runspaces debuggen .....	271
14.7	Zusammenfassung .....	273
<b>15</b>	<b>Sicherheit .....</b>	<b>275</b>
15.1	Die Rolle der Ausführungsrichtlinie .....	275
15.2	Umgang mit Credentials .....	276
15.2.1	Einen Secure String lesbar machen .....	278
15.2.2	Einen Secure String anlegen .....	279
15.2.3	Secure String-Dateien sicher speichern .....	279
15.3	Umgang mit Zertifikaten .....	280
15.4	Weiterer Umgang mit Zertifikaten .....	281
15.4.1	Auflisten bestimmter Zertifikate .....	281
15.5	Zertifikate anlegen .....	281
15.5.1	Selbstsignierte Zertifikate erstellen .....	282
15.6	Skripte signieren .....	285
15.7	Zeichenketten verschlüsseln .....	286
15.7.1	Texte verschlüsseln und entschlüsseln mit CipherNet .....	286
15.7.2	Zeichenketten mit Zertifikaten verschlüsseln .....	287
15.8	Umgang mit Zugriffsberechtigungen .....	288
15.8.1	Entfernen von Zugriffsberechtigungen .....	290
15.9	Die Befehlsausführung protokollieren .....	291
15.9.1	Befehlsprotokollierung per Gruppenrichtlinien steuern .....	291
15.9.2	Mehr Möglichkeiten bei Start-Transcript .....	293

---

15.10	PowerShell-Remoting-Endpunkte sichern mit Just Enough Administration (JEA) . . . . .	294
15.10.1	JEA in der Praxis . . . . .	295
15.11	Eine Session ohne Remoting einschränken . . . . .	298
15.12	Das Invoke-Expression-Cmdlet und warum es potentiell gefährlich ist . . .	299
15.12.1	Invoke-Expression per Scriptblock-Logging überwachen . . . . .	300
15.13	Zusammenfassung . . . . .	301
<b>16</b>	<b>PowerShell für Linux . . . . .</b>	<b>303</b>
16.1	Ein erster Überblick . . . . .	303
16.2	PowerShell unter Linux installieren . . . . .	304
16.3	Die ersten Schritte unter Linux . . . . .	305
16.4	Navigieren im Dateisystem . . . . .	305
16.5	PowerShell-Remoting mit SSH . . . . .	305
16.6	PowerShell versus PowerShell Core . . . . .	307
16.7	Zusammenfassung . . . . .	307
<b>17</b>	<b>Wie man gute Skripte schreibt . . . . .</b>	<b>309</b>
17.1	Kommentare . . . . .	309
17.2	Variableninitialisierung erzwingen . . . . .	309
17.3	Aliase vermeiden . . . . .	310
17.4	Datentypen für Parameter . . . . .	310
17.5	Keine „Spuren“ hinterlassen . . . . .	310
17.6	Der PSScriptAnalyzer . . . . .	311
17.6.1	Eigene Regeln definieren . . . . .	311
17.7	Zusammenfassung . . . . .	313
<b>18</b>	<b>Spaß mit der PowerShell . . . . .</b>	<b>315</b>
18.1	Zufallszahlen . . . . .	315
18.2	Farbige Ausgaben . . . . .	317
18.2.1	Farbe in der Konsole dank VT100-Unterstützung . . . . .	318
18.3	Ein etwas anderer Prompt . . . . .	320
18.3.1	Ein farbiger Prompt . . . . .	321
18.4	Sounddateien abspielen . . . . .	322
18.4.1	Systemso unds abspielen . . . . .	322
18.4.2	Töne erzeugen . . . . .	323
18.4.3	PowerShell-Musik . . . . .	325
18.5	Die PowerShell lernt sprechen . . . . .	325
18.5.1	Die .NET-Laufzeit kann auch sprechen . . . . .	326
18.6	ASCII-Art und die 80er-Jahre . . . . .	327
18.6.1	Bewegte ASCII-Art . . . . .	327
18.7	Ein Zitat, bitte . . . . .	328
18.7.1	Einen Internet-Zeitserver abfragen . . . . .	329



---

18.8	Ein Matrix-Style-Bildschirmschoner . . . . .	330
18.9	HAL ist IBM – der unwiderlegbare Beweis . . . . .	330
18.10	April, April. . . . .	330
18.11	Ein Spielhallenklassiker. . . . .	332
18.12	Zusammenfassung . . . . .	332
<b>19</b>	<b>PowerShell für Entwickler . . . . .</b>	<b>333</b>
19.1	Unterschiede und Gemeinsamkeiten mit C# . . . . .	333
19.2	Umgang mit Assemblys. . . . .	334
19.3	Assemblys in eine PowerShell-Sitzung laden . . . . .	335
19.3.1	Assemblys über ihren Pfad laden . . . . .	336
19.3.2	Assemblys über ihren Namen laden . . . . .	336
19.3.3	Alle geladenen Assemblys auflisten . . . . .	337
19.3.4	Klassendefinitionen sichtbar machen . . . . .	338
19.3.5	Den Inhalt einer Assembly sichtbar machen . . . . .	338
19.4	Assemblys erstellen . . . . .	340
19.4.1	Herunterladen von NuGet-Packages . . . . .	341
19.5	Umgang mit generischen Typen . . . . .	341
19.5.1	Generische Listen . . . . .	342
19.5.2	Generische Methodenaufrufe . . . . .	342
19.5.3	Aufruf einer generischen privaten Methode . . . . .	343
19.6	Umgang mit Events . . . . .	344
19.7	Das erweiterbare Typensystem . . . . .	346
19.8	Cmdlets definieren . . . . .	347
19.9	Benutzeroberflächen mit WPF. . . . .	349
19.10	Win32-API-Funktionen aufrufen. . . . .	353
19.11	Zusammenfassung . . . . .	354
	<b>Glossar . . . . .</b>	<b>355</b>
	<b>Stichwortverzeichnis. . . . .</b>	<b>359</b>

---

## Zusammenfassung

Dieses Kapitel gibt eine kompakte Einführung in die Grundlagen der PowerShell für alle Leser, die die PowerShell zwar kennen, aber mit diesen Grundlagen noch nicht in allen Details vertraut sind. Außerdem werden in diesem Kapitel die wichtigsten Begriffe vorgestellt, die den theoretischen Unterbau der PowerShell-Infrastruktur betreffen.

---

## 1.1 Das WMF im Überblick

Das *Windows Management Framework* (WMF) ist der Überbau, der nicht nur die beiden PowerShell-Hostanwendungen PowerShell-Konsole und PowerShell ISE, sondern auch eine Reihe von Komponenten (in Gestalt von Assembly-Dateien) umfasst, die für die Ausführung von PowerShell-Funktionalitäten eine Rolle spielen. Die aktuelle Version ist WMF 5.1. Wenn diese Version nicht bereits Teil des Betriebssystems ist, wie bei Windows Server 2016 und Windows 10 Anniversary Update (Version 1607), kann es als Update nachträglich installiert werden.<sup>1</sup> Da sich Microsoft dazu entschieden hat, das WMF für alle aktuellen Windows-Versionen anzubieten, ist es kein Problem, WMF 5.1 unter Windows Server 2008 R2 und Windows 7 zu installieren. Im Unterschied zur Installation von WMF 5.0 ist es bei einem Update von den Versionen 2.0 und 3.0 nicht erforderlich, dass zuerst WMF 4.0 als Zwischenschritt installiert wird. Tab. 1.1 stellt die Bestandteile von WMF 5.1 zusammen. Abb. 1.1 zeigt die Bestandteile des WMF in einem Schaubild.

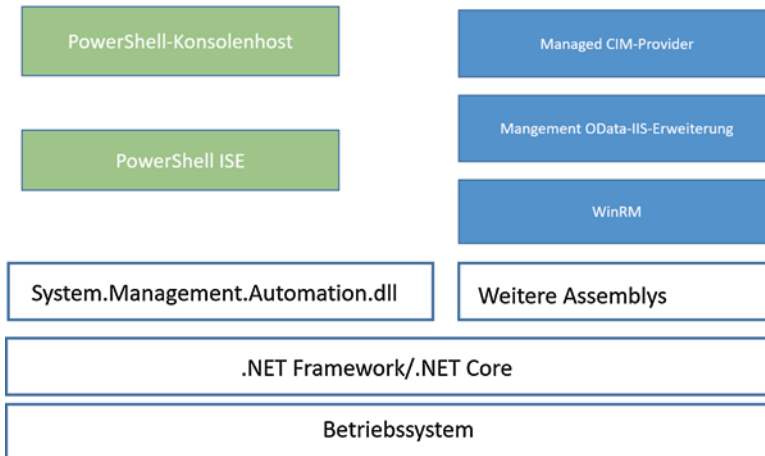
---

<sup>1</sup> Auch wenn sich solche Downloadlinks mit der favorisierten Suchmaschine im Allgemeinen schnell finden lassen und Erweiterungen in der Regel über die PowerShell Gallery hinzugefügt werden, habe ich in meinem Blog <http://poshadmin.de> die Links der wichtigsten PowerShell-Downloads zusammengestellt.

**Tab. 1.1** Die Komponenten des Windows Management Frameworks

Komponente	Bedeutung
Windows-PowerShell	Zwei Host-Anwendungen für die interaktive Eingabe von Befehlen und das Ausführen von Skripten.
DSC	Über die <i>Desired State Configuration</i> ist das Übertragen von Konfigurationsänderungen auf der Grundlage einer Beschreibungssprache möglich.
WinRM	Ermöglicht das Ausführen von Befehlen auf anderen Computern im Netzwerk auf der Grundlage von Ws-Management.
Managed WMI	Implementierung von WMI/CIM auf der Grundlage der .NET-Laufzeit.
PowerShell Webdienste	Ermöglicht das Ausführen von Cmdlets über REST-basierte Webservice-Aufrufen im Zusammenspiel mit der IIS-Erweiterung und Management OData.
SIL	<i>Software Inventory Logging</i> . SIL wurde mit Windows Server 2012 R2 vorgestellt.
CIM-Provider	Ein CIM/WMI-Provider umfasst die Definitionen von CIM-Klassen in Gestalt von MOF-Dateien. Traditionell basieren Provider auf COM-Komponenten. WMF ermöglicht es, dass ein CIM-Provider auch in der Programmiersprache C umgesetzt werden kann. Damit wird der Aufwand für das Erstellen eines Providers für die Hersteller von Hardwarekomponenten deutlich reduziert und die Provider werden plattformunabhängig.

## Das Windows Management Framework (WMF) im Überblick

**Abb. 1.1** Das WMF im Überblick

- **Tipp** Wie sich WMF 5.1 per SCCM verteilen lässt, wird in einem Blog-Eintrag von *Anders Rodland* anschaulich beschrieben: <https://www.andersrodland.com/deploy-windows-management-framework-51-with-sccm>.

---

## 1.2 Die .NET-Laufzeit

Die .NET-Laufzeit, auch .NET Framework oder einfach nur .Net (ausgesprochen als „dot-net“), ist eine Laufzeitumgebung für Anwendungen und der Unterbau der PowerShell. Sie ist nicht Teil des WMF, sondern muss separat installiert werden. Windows besitzt ab Windows Server 2008 R2 und Windows 7 zwar von Anfang an die .NET-Laufzeit, oft aber nicht in der passenden Version. WMF 5.1 setzt bereits .NET 4.6.1 voraus. Die .NET-Laufzeit ist nicht für die Ausführung der PowerShell-Assemblies zuständig, sondern bietet eine Fülle von Funktionalitäten, die natürlich direkt in der PowerShell-Konsole oder in einem Skript verwendet werden können. Dazu gleich eine kleine Kostprobe. Der folgende Befehl gibt über einen true/false-Wert an, ob der PowerShell-Host als Administrator gestartet wurde:

```
[System.Security.Principal.WindowsIdentity]::GetCurrent().Groups.Value -  
Contains "S-1-5-32-544"
```

Keine Lust, ein solches „Befehlsmonster“ abzutippen? Kein Problem. Zum einen gibt es in der PowerShell-Konsole auch für solche Befehle eine Eingabevervollständigung per [Tab]-Taste, so dass das Eintippen relativ schnell geht. Zum anderen gibt es Erweiterungen wie das *Carbon*-Modul, das bereits fertige Befehle dafür enthält. Wo gibt es das *Carbon*-Modul? Zum Beispiel in der PowerShell Gallery. Und wie erhalte ich das Modul? Ganz einfach über das *Install-Module*-Cmdlet der PowerShell.

Der folgende Befehl fügt das *Carbon*-Modul von der PowerShell Gallery hinzu:

```
Install-Module -Name Carbon -Scope CurrentUser -Force
```

Der *Scope*-Parameter sorgt dafür, dass das Modulverzeichnis nicht im Programm-Verzeichnis (dazu müsste die PowerShell als Administrator gestartet werden), sondern im Dokumente-Verzeichnis abgelegt wird. Anschließend steht der Befehl „Test-AdminPrivilege“ zur Verfügung, der einen true-Wert zurückgibt, wenn die PowerShell-Hostanwendung mit Administratorberechtigungen gestartet wurde.

### 1.2.1 Assemblies

Der Begriff Assembly (engl. für Versammlung) steht bei der .NET-Laufzeit lediglich für eine Datei, die „Managed Code“ enthält. Managed Code wiederum ist ein Befehlscode, der von der *Common Language Runtime* (CLR) ausgeführt wird. Die CLR ist ein

Kernbestandteil der .NET-Laufzeit. Die PowerShell besteht aus einer Reihe solcher Assembly-Dateien. Die wichtigste ist *System.Management.Automation.dll*. Diese Dateien sind keine abstrakten Größen, sondern liegen auf der Festplatte des Computers. Der folgende PowerShell-Befehl gibt die Pfade der aktuell geladenen Assembly-Dateien aus:

```
[AppDomain]::GetAssemblies().CurrentDomain.Location
```

Eine Assembly(-Datei) enthält in der Regel eine Vielzahl von Typdefinitionen, in der Regel in Gestalt von Klassendefinitionen. Jede Klasse definiert die Members eines Objekts. Jedes PowerShell-Cmdlet basiert auf einer Klassendefinition. Die Klassendefinition selber lässt sich nicht so ohne weiteres ausgeben (dazu wird ein sogenannter IL-Disassembler wie zum Beispiel das Programm *ILSpy* benötigt), wohl aber die Members und die genaue Bezeichnung der Klassendefinition.

Der folgende Befehl gibt den Namen der Klassendefinition und die Members jener Klasse aus, auf der das *Get-Command*-Cmdlet basiert:

```
Get-Command -Name Get-Command | Get-Member
```

Zuerst holt das *Get-Command*-Cmdlet das *Get-Command*-Cmdlet als Objekt (daher kommt „Get-Command“ zwei Mal vor). Das daraus resultierende *CmdletInfo*-Objekt wird per Pipe-Operator dem *Get-Member*-Cmdlet übergeben, das dann die Members des Objekts ausgibt.

Die besondere Bedeutung von Assemblys für die PowerShell besteht darin, dass eine Assembly-Datei in der Regel viele Definitionen von Klassen, Schnittstellen und Konstantenlisten enthält, die allgemein unter dem Begriff „Typen“ zusammengefasst werden.

Es ist faszinierend, wie einfach sich zum Beispiel die Klassendefinitionen einer Assembly auflisten lassen. Dafür ist die unscheinbare Eigenschaft *Assembly* zuständig, die es bei jedem Typobjekt gibt. Sie liefert einen Verweis auf jenes Objekt, das die Assembly-Datei repräsentiert, in der der Typ definiert ist.

### Beispiel

Der folgende Befehl gibt die Namen aller öffentlichen Klassen aus, die in der PowerShell-Assembly *System.Management.Automation.dll* enthalten sind:

```
[PSObject].Assembly.GetTypes() | Where { $_.IsPublic -and $_.IsClass } |
Select Namespace, Name
```

*PSObject* ist der Name einer Klasse, die in *System.Management.Automation.dll* definiert ist. Würden Sie stattdessen „Object“ schreiben, würde die *Assembly*-Eigenschaft eine andere Assembly ansprechen und eine ganz andere Ausgabe wäre die Folge. Eine kleine Änderung mit großer Wirkung. Mehr über den Umgang mit Assemblys erfahren Sie im Anhang dieses Buches, in dem die PowerShell aus der Perspektive eines Software-Entwicklers vorgestellt wird.

## 1.3 Die Objekt-Pipeline

Das sicherlich wichtigste Unterscheidungsmerkmal zwischen der PowerShell und anderen Shells wie der *Bash*, die seit *Windows 10 Anniversary Update* ein optionaler Bestandteil des Betriebssystems ist, dem *Windows Scripting Host* oder den guten alten Stapeldateien ist der Umstand, dass bei der PowerShell ausschließlich Objekte über die Pipeline übertragen werden. Ein Objekt beschreibt einen Gegenstand wie einen Prozess oder einen Systemdienst und stellt die Details über diesen Gegenstand über Namen zur Verfügung, die allgemein Properties (engl. für Eigenschaften) heißen. Eine Property ist nicht nur ein Name, sondern immer mit einem Datentyp verbunden, so dass die Information darüber, wie der Wert behandelt werden muss, in die Eigenschaft eingebaut ist. Das bedeutet konkret, dass die Weiterverarbeitung einer Ausgabe sehr einfach wird. Sollen zum Beispiel die laufenden VMs nach ihrer Laufzeit sortiert werden, erledigt dies eine Kombination aus *Get-VM* und *Sort-Object*:

```
Get-VM | Sort-Object Uptime
```

„Uptime“ ist der Name einer der zahlreichen Eigenschaften, die ein Objekt besitzt, das von *Get-VM* in die Pipeline gelegt wird, und die für die bisherige „Uptime“ der VM steht. Da dieser Wert nicht einfach eine Zahlfolge oder eine Zeichenkette, sondern selber ein Objekt vom Typ *TimeSpan* ist, wird der Wert jedes einzelnen Objekts so interpretiert, dass daraus automatisch die richtige Reihenfolge resultiert.

Während in diesem Fall ein Wert wie 11:45:00, der für eine Laufzeit von 11 Stunden, 45 Minuten und 0 Sekunden steht, auch als Zeichenfolge passend interpretiert werden würde, sieht dies bei einem Datumszeitwert etwas anders aus. Hier ist der Wert „1/1/2017“ immer dann kleiner als der Wert „2/1/2007“, wenn beide Werte als Zeichenfolge verglichen würden. Da die Eigenschaft *StartTime* bei einem Prozessobjekt vom Typ *DateTime* ist, kann auch hier das *Sort-Object*-Cmdlet den Inhalt der Pipeline so sortieren wie es einem Datumszeitwert entspricht:

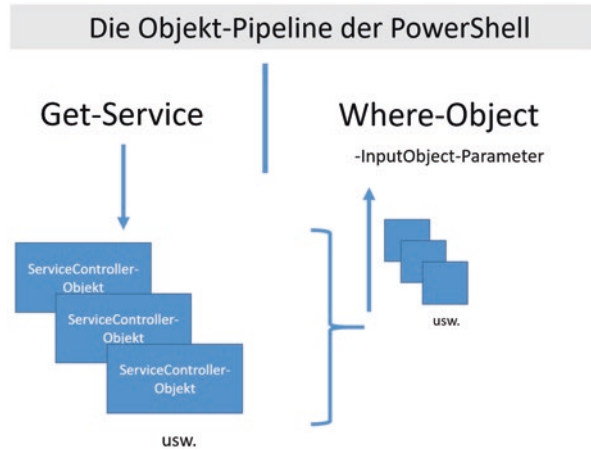
```
Get-Process | Sort-Object StartTime
```

Um es salopp zu formulieren: Dem *Sort-Object* muss nicht gesagt werden, wie es die Werte einer Eigenschaft sortieren muss. Es erfährt dies aus dem Typ der jeweiligen Eigenschaft, in dem die Sortierfähigkeit „eingebaut“ ist. Wäre die Rückgabe eines Cmdlets lediglich Text, wäre ein Sortieren der Ausgabe deutlich aufwändiger. Abb. 1.2 veranschaulicht das Prinzip der Objekt-Pipeline.

Nicht nur das Sortieren profitiert davon, dass über die Pipeline Objekte übergeben werden. Jede Operation wird deutlich einfacher oder überhaupt erst mit vertretbarem Aufwand möglich. Sollen die Eckdaten aller Prozesse, die aktuell mehr als 100MB im Arbeitsspeicher belegen, sortiert nach dem Wert der Arbeitsspeicherbelegung im HTML-Format ausgegeben werden, erledigt das der folgende Befehl:

```
Get-Process | Where-Object WS -gt 100MB | Sort-Object WS -Descending |  
Select-Object Name, WS, StartTime | ConvertTo-Html
```

**Abb. 1.2** Das Prinzip der Objekt-Pipeline – die Ausgabe eines Cmdlets wird in Gestalt von Objekten an den Parameter eines anderen Cmdlets gebunden



Soll der Output in einer Textdatei gespeichert werden, muss die Ausgabe entweder per `>` umgeleitet oder per `Out-File`-Cmdlet gespeichert werden:

```
Get-Process | Where-Object WS -gt 100MB | Sort-Object WS -Descending |
Select-Object Name, WS, StartTime | ConvertTo-Html > Prozesse.htm
```

## 1.4 PSProvider und PSDrives

Ein weitere Innovation bei der PowerShell war bei ihrer Einführung im Jahr 2006 der Umstand, dass der Laufwerksbegriff neu definiert wurde. Ein PSDrive ist bei der PowerShell ein virtuelles Laufwerk, über das beliebige Ablagen einheitlich, also mit demselben Satz an Cmdlets, angesprochen werden. Das `Get-PSDrive`-Cmdlet listet alle Laufwerke auf. Neben den Dateisystemlaufwerken gibt es unter anderem die PSDrives `Function` für die PowerShell-Functions, `Variable` für die PowerShell-Variablen, `env` für die Umgebungsvariablen des PowerShell-Prozesses und `cert` für die lokalen Zertifikate. Damit listet ein „dir C:“ das Stammverzeichnis von Laufwerk C:, ein „dir env:“ die Umgebungsvariablen des Prozesses auf. Ein „Get-Content – Path C:\Windows\Win.ini“ gibt den Inhalt der angesprochenen Datei, ein „Get-Content – Path Function:Get-FileHash“ den Inhalt der angesprochenen Function-Definition aus. Was genau geholt wird, legen stets der `Path`-Parameter des Cmdlets fest und das Laufwerk, das über den Pfad angesprochen wird.

Alle PSDrive-Laufwerke werden durch PSProvider zur Verfügung gestellt. Das `Get-PSProvider`-Cmdlet listet die aktuell geladenen PSProvider auf. Über das Importieren von Modulen kommen weitere PSProvider hinzu. Ein Beispiel ist das `ActiveDirectory`-Modul, das den gleichnamigen PSProvider lädt, der ein Laufwerk mit dem Namen „AD“ hinzufügt. Über das AD-Laufwerk kann das Active Directory-Verzeichnis wie ein Laufwerk angesprochen werden. Ein „dir AD: -Recurse“ gibt den gesamten Inhalt des Verzeichnisses aus.

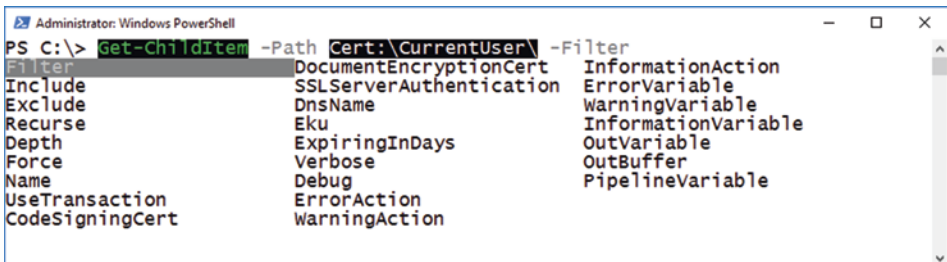
### 1.4.1 Dynamische Parameter

Da ein Cmdlet wie *Get-ChildItem* für jeden Laufwerkstyp passen muss, umfasst seine Parameterliste offiziell nur Parameter, die für sämtliche Laufwerkstypen geeignet sind. Das bedeutet im Umkehrschluss, dass es Parameter gibt, die nur auf bestimmte Laufwerkstypen angewendet werden können. Ob zum Beispiel die Parameter *Filter* oder *Recurse* bei einem Laufwerk verwendet werden können, hängt davon ab, ob diese Option durch den PSProvider, der das Laufwerk zur Verfügung stellt, implementiert wurde.

Damit Parameter in Abhängigkeit des Laufwerkstyps zur Verfügung stehen und z. B. das *Get-ChildItem*-Cmdlet je nach Laufwerkstyp, der über den *Path*-Parameter angegeben wird, einen *Recurse*-Parameter besitzt oder nicht, gibt es die dynamischen Parameter. Die dynamischen Parameter sind Teil des Providers und stehen immer dann zur Verfügung, wenn über den *Path*-Parameter des Cmdlets ein Laufwerk angesprochen wird, das von diesem PSProvider zur Verfügung gestellt wird. Ein Beispiel ist der Parameter *Directory*, den es beim *Get-ChildItem*-Cmdlet nur dann gibt, wenn ein hierarchisches Laufwerk angesprochen wird. Den Parameter *CodeSigningCert* gibt es nur dann, wenn über den *Path*-Parameter das *Cert*-Laufwerk angesprochen wird.

In der PowerShell-Hilfe werden die dynamischen Parameter nicht in der Hilfe zu dem jeweiligen Cmdlet, sondern in jeder Hilfe zu dem jeweiligen Provider beschrieben. Ein „Help Certificate“ listet zum Beispiel die Beschreibung des *Cert*-Providers auf, in der auch alle dynamischen Parameter enthalten sind.

- **Tip** Eine Übersicht über alle zur Auswahl stehenden Parameter liefert die Tastenkombination [Strg]+[Leertaste] (sofern das *PSReadline*-Modul geladen ist). Diese richtet sich danach, welches Laufwerk über den *Path*-Parameter ausgewählt wurde. Abb. 1.3 zeigt die Parameterausgabe beim *Get-ChildItem*-Cmdlet, wenn über den *Path*-Parameter des Cmdlets das *Cert*-Laufwerk angesprochen wird.

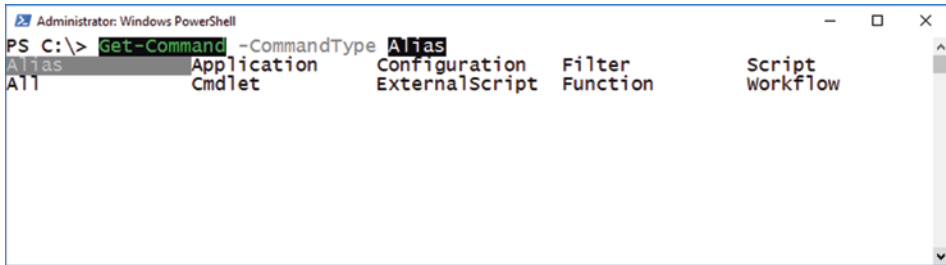


```

Administrator: Windows PowerShell
PS C:\> Get-ChildItem -Path Cert:\CurrentUser -Filter
Filter DocumentEncryptionCert InformationAction
Include SslServerAuthentication ErrorVariable
Exclude DnsName WarningVariable
Recurse Eku InformationVariable
Depth ExpiringInDays OutVariable
Force Verbose OutBuffer
Name Debug PipelineVariable
UseTransaction ErrorAction
CodeSigningCert WarningAction
  
```

**Abb. 1.3** Dank PSReadLine werden alle Parameter eines Cmdlets inklusive der dynamischen Parameter übersichtlich angezeigt





**Abb. 1.4** Dank PSReadline werden auch die Parameterwerte eines Parameters übersichtlich angezeigt

## 1.5 Functions, Aliase, Workflows und Configurations

Cmdlets sind nicht der einzige Typ von Kommandos, der bei der PowerShell zur Auswahl steht. Darüber hinaus gibt es Functions, Aliase, Workflows, Skripte und Konfigurationen. Das *Get-Command*-Cmdlet zeigt per Voreinstellung nur Cmdlets, Aliase und Functions an. Möchte man weitere oder alle Commandtypen sehen, gibt es dafür den Parameter *CommandType*, dem die Namen der Commandtypen per Komma getrennt übergeben werden. In der Regel sind dies *Alias*, *Cmdlet*, *Function* und *Application*. Die Ausgabe kann dabei so umfangreich werden, dass in der Konsole die Puffertiefe auf den Maximalwert 9999 gesetzt werden muss.<sup>2</sup> Der Commandtyp *Application* gibt alle Programmdateien zurück, die sich in einem der Verzeichnisse befinden, die Teil der *Path*-Umgebungsvariablen sind. Dazu zählen nicht nur Exe-Dateien, sondern auch Dateien mit den Erweiterungen *.Cmd* und *.Bat*.

- **Tip** Auch bei der Anzeige der möglichen Werte für einen Parameter bietet das *PSReadline*-Modul eine praktische Eingabehilfe an. Nach der Eingabe des Parameternamens und eines Leerzeichens zeigt ein [Strg]+[Leerzeichen] die für den Parameter zur Auswahl stehenden Werte an. Abb. 1.4 zeigt eine *PsReadline*-Ausgabe für den *CommandType*-Parameter von *Get-Command*.

## 1.6 Erweiterbarkeit

Ein weiterer Pluspunkt der PowerShell ist ihre Erweiterbarkeit. Eine Erweiterung ist entweder ein Snap-In oder ein Modul. Snap-Ins spielen als Erweiterungen nur noch in Ausnahmefällen eine Rolle. Eine dieser Ausnahmen sind die Cmdlets aus dem *Xen App SDK* von *Citrix*. Es wird noch weitere Ausnahmen geben. Die typische PowerShell-Erweiterung liegt in Gestalt eines Moduls vor. Ein Modul basiert auf einem Verzeichnis, das eine oder

<sup>2</sup> Auf meinem Windows 10-PC sind es 8079 Commands. Wer bietet mehr?

mehrere PowerShell-Dateien enthält. Damit ein Verzeichnis als Modulverzeichnis erkannt wird, muss es eine Psd1-, Psm1- oder eine Dll-Datei enthalten, deren Name dem Modulverzeichnis entspricht. Spielt die Versionsnummer des Moduls eine Rolle, enthält das Modulverzeichnis auf der obersten Ebene ein Verzeichnis mit der Versionsnummer als Name, in dem sich die Psd1- oder Psm1-Datei befindet. Damit können mehrere Versionen eines Moduls parallel vorliegen. Alle „offiziellen“ Modulverzeichnisse sind in der Umgebungsvariable *PSModulePath* enthalten. Dazu zählen *\$PSHome\Modules*, *\$Home\Documents\WindowsPowerShell\Modules* und *\$env:ProgramFiles\WindowsPowerShell\Modules*. Auf einem 64-Bit-Windows kommt *\$env:ProgramFiles(x86)\WindowsPowerShell\Modules* als weiteres Modulverzeichnis hinzu, in dem sich die Modulverzeichnisse für 32-Bit-PowerShell-Hosts befinden. Grundsätzlich spielt es keine Rolle, in welchem Verzeichnis sich ein Modulverzeichnis befindet. Modulverzeichnisse, die in keinem der offiziellen Modulverzeichnisse liegen, müssen per *Import-Module* direkt geladen werden. Alle anderen Module werden dadurch geladen, dass ein Command (Cmdlet oder Function) ausgeführt wird, das in dem Modul definiert wird. Die PowerShell legt im Benutzerprofil einen Cache an, in dem die Namen aller Commands in allen verfügbaren Modulen enthalten sind, so dass ein Modul relativ schnell nach Eingabe eines Commandnamens lokalisiert wird.

---

**Beispiel**

Der folgende Befehl lädt das Modul *PsKurs*, das sich im Verzeichnis *C:\MeineModule* befindet:

```
Import-Module -Name C:\MeineModule\PsKurs
```

Die Psd1- oder Psm1-Datei muss beziehungsweise soll nicht explizit angegeben werden. Befinden sich in dem Verzeichnis mehrere Versionen des Moduls, wird automatisch die aktuellste Version geladen. Ansonsten muss die Version über den Parameter *RequiredVersion* explizit angefordert werden.

---

**Beispiel**

Der folgende Befehl importiert die Version 1.1 des Moduls *PsKurs*:

```
Import-Module -Name C:\MeineModule\PsKurs -RequiredVersion 1.1
```

Ein im Praxisalltag nicht selten vorkommendes Missverständnis besteht darin, dass davon ausgegangen wird, dass ein *Get-Module* alle verfügbaren Module auflistet. Das ist aber nicht der Fall. Das *Get-Module*-Cmdlet zeigt nur die aktuell geladenen Module an. Möchte man alle verfügbaren Module sehen, muss der Parameter *ListAvailable* angehängt werden. Auch hier kann ein Missverständnis auftreten. Da die Module nach Verzeichnissen gruppiert ausgegeben werden, kann es passieren, dass man ein Modul nicht entdeckt, da es sich im Benutzerprofilverzeichnis befindet und daher als erstes ausgegeben wurde.

Möchte man die Modulliste etwas übersichtlicher erhalten, muss ein *Select-Object* angehängt werden:

```
Get-Module -ListAvailable | Select Name, Version, ModuleType | Sort-Object Name
```

Was auch nicht jeder erfahrenere PowerShell-Anwender weiß: Die Ausgabe in der Konsole ist nicht in Stein gemeißelt, über das *Out-GridView*-Cmdlet kann jede Ausgabe in einem Fenster angezeigt werden:

```
Get-Module -ListAvailable | Select Name, Version, ModuleType | Sort-Object Name | Out-GridView
```

Jetzt sieht die Ausgabe doch gleich etwas freundlicher aus.

Und es gibt noch einen weiteren Grund für Missverständnisse. Unter einer 32-Bit-PowerShell werden unter einem 64-Bit-Windows nur die Verzeichnisse in *C:\Program Files(86)\WindowsPowerShell\Modules* berücksichtigt, nicht jene, die sich in *C:\Program Files\WindowsPowerShell\Modules* befinden und umgekehrt.

Was macht *Get-InstalledModule*? Die Function aus dem *PowerShellGet*-Modul listet nur jene Module auf, die über das *Install-Module*-Cmdlet hinzugefügt wurden.

### 1.6.1 Ein Modulmanager in Gestalt des PowerShellGet-Moduls

Die Zeiten, in denen ein Modul als Zip-Datei von der Webseite des Modulautors heruntergeladen, ausgepackt und der Inhalt in ein Verzeichnis kopiert werden musste, sind seit der Version 5.0 vorbei. Seit dieser Version umfasst die PowerShell offiziell einen „Paketmanager“, der auch das Laden von Modulen übernimmt.<sup>3</sup> Die Befehle für die Modulverwaltung enthält das Modul *PowerShellGet*. Für das Aufspüren von Modulen gibt es die Function *Find-Module*. Sie durchsucht die von Microsoft betriebene PowerShell Gallery (PSGallery) unter <http://powershellgallery.com>.

#### Beispiel

Der folgende Befehl gibt alle Module aus, in deren Namen das Wort „Excel“ enthalten ist:

```
Find-Module *excel*
```

Beim ersten Aufruf werden Sie aufgefordert, Provider für den NuGet-Paketmanager zu installieren, über den die Zugriffe abgewickelt werden. Per *Install-Module* wird ein Modul lokal hinzugefügt. Je nach Wert für den *Scope*-Parameter wird das Modulverzeichnis entweder unter *C:\Programfiles\WindowsPowerShell\Modules* oder unter *\$Home\Documents\WindowsPowerShell\Modules* abgelegt.

<sup>3</sup>Für die Versionen 3.0 und 4.0 steht der Paketmanager-Manager ebenfalls zur Verfügung, aktuell (Stand: März 2017) aber immer noch in Gestalt der „Package Management Preview“.

Über *Get-InstalledModule* werden alle per PowerShellGet installierten Module aufgelistet. Auch Skripte lassen sich auf diese Weise von der PowerShell Gallery laden. Ein „Find-Script \*“ gibt alle verfügbaren Skripte aus, per *Install-Script* wird ein Skript lokal abgelegt.

So elegant das Hinzufügen von Modulen und Skripten über die PowerShell Gallery ist, möchte und kann nicht jeder Administrator fremden Programmcode laden. Auch wenn die Wahrscheinlichkeit sehr gering ist, kann es nicht ausgeschlossen werden, dass ein Modul oder Skript „Schadcode“ enthält. Anders als man es vermuten würde, wird ein von einem grundsätzlich anonymen Nutzer hochgeladenes Modul nicht geprüft. Es steht unmittelbar nach dem Upload zum Download zur Verfügung. Das PowerShell-Team beabsichtigt zwar die Einführung von „High Quality-Modulen“, die geprüft, versioniert und signiert sind, doch wird es eine Weile dauern bis sich dieser Modultyp verbreitet. Die Alternative zur öffentlichen PowerShell Gallery besteht aktuell darin, ein Repository selber zu hosten, entweder im Internet, zum Beispiel unter Azure, oder im Intranet. In diesem Repository werden nur ausgewählte Module abgelegt. Eine Option ist die „Private PSGallery“, die Microsoft als GitHub-Projekt zur Verfügung stellt. Mehr zu diesem Thema in Kap. 7, in dem es um das Bereitstellen von Skripten und Modulen geht.

---

## 1.7 Befehlssyntax

Die Syntax der PowerShell-Befehle wirkt auf einen Anfänger nicht gerade konsistent. Das betrifft auch die Ausgabe eines Befehls. Dabei war die Konsistenz eines der obersten Ziele bei der Planung der PowerShell gewesen. Ziel verfehlt? Das natürlich nicht. Man muss nur wissen, dass es bei der PowerShell-Syntax verschiedene Abkürzungen gibt:

- Für Namen eines Command können Aliase verwendet werden.
- Ist ein Parameter ein Positionsparameter, kann sein Name entfallen.
- Auch für Parameter gibt es Aliase (zum Beispiel „ea“ für den Parameter *ErrorAction*).
- Generell kann der Name eines Parameters so stark verkürzt werden, dass er noch eindeutig ist in Bezug auf die übrigen Parameter des Commands. Der Parameter *ForegroundColor* beim *Write-Host*-Cmdlet kann daher durch „Fore“, „Fo“ und sogar „f“ abgekürzt werden. Dass sich dadurch eine bunte Mischung unterschiedlicher Schreibweisen ergibt, wirkt bei oberflächlicher Betrachtung natürlich nicht gerade konsistent. Dahinter stecken aber einfache Namensregeln.
- Diese Besonderheit dürfte auch so mancher erfahrene PowerShell-Admin nicht kennen. Bereits seit der Version 1.0 kann bei den Get-Cmdlets das Get-Verb weggelassen werden. Ein „Service“ gibt daher die Eckdaten zu allen Systemdiensten aus, ein „localuser“ die Eckdaten zu allen lokalen Benutzerkonten. Ein „Process“ gibt allerdings nicht die Eckdaten zu allen Prozessen aus, da „Process“ ein reserviertes Wort ist.

---

**Beispiel**

Die folgenden Befehle sind identisch bezüglich ihrer Syntax:

```
Get-Process -Name Svchost | Select-Object -Property StartTime, Id, Ws
Get-Process Svchost | Select StartTime, Id, Ws
gps Svchost | Select StartTime, Id, Ws
```

Name und Property sind jeweils Positionsparameter. Sie erhalten ihren Wert daher auch aufgrund der Position eines Wertes, dem kein Parametername vorausgeht. Ob ein Parameter ein Positionsparameter ist, erfährt man aus der Hilfe zu dem Parameter. Mit anderen Worten: Auch wenn der Name eines Parameters entfallen kann, ist der Parameter trotzdem im Spiel.

Während innerhalb von Skripten Aliase aufgrund der Lesbarkeit von Befehlen nicht verwendet werden sollten, spricht (natürlich) nichts gegen die Verwendung von Aliasen in der Konsole. Im Gegenteil, je kürzer desto besser.

---

## 1.8 Die moderne Konsole

Viele Anwender, die die PowerShell unter Windows Server 2016 oder Windows 10 starten, denken sich: Toll, die Konsole ist farbig geworden und selbstverständliche Kleinigkeiten wie ein Kopieren und Einfügen von Texten über [Strg]+[C] und [Strg]+[V] sind (endlich) möglich. Sie installieren die PowerShell 5.0 oder 5.1 unter Windows Server 2012 und stellen fest: Alles sieht aus wie immer und [Strg]+[V] fügt nichts ein. Der Grund für die auf Anhieb nicht ganz nachvollziehbare Diskrepanz ist einfach: Ab Windows Server 2016 und Windows 10 wird automatisch das Modul *PSReadline* geladen, das unter älteren Windows-Versionen zunächst hinzugefügt werden muss. *PSReadline* ist ein Modul, das den Standard-Befehlszeileneditor der Konsole durch eine neue Version, die deutlich mehr kann, ersetzt. Dazu gehören unter anderem eine Fülle von Tastaturshortcuts, die an den Unix-Editor EMACS angelehnt wurden, neue Möglichkeiten wie das Markieren der kompletten Befehlszeile über ein vertrautes [Strg]+[A], das Kopieren und Einfügen über [Strg]+[C] und [Strg]+[V], eine automatische Syntaxkontrolle während der Eingabe mit einer Fehleranzeige in Gestalt eines roten Promptzeichens und eben eine Syntaxeinfärbung. Der größte Komfortgewinn, der mit *PSReadline* einhergeht, ist der Umstand, dass sich in der Konsole auch eine mehrzeilige Eingabe komfortabel editieren lassen (siehe Abb. 1.5).

*PSReadline* setzt lediglich die Version 3.0 der PowerShell voraus, so dass grundsätzlich nichts dagegen sprechen sollte, es bei jeder PowerShell-Installation hinzuzufügen. Aber wie? Am einfachsten natürlich über ein *Install-Module*. Der folgende Befehl setzt voraus, dass die PowerShell-Konsole als Administrator gestartet wurde:

```
Install-Module PSReadline -Force
```

```

Administrator: Windows PowerShell
PS C:\> get-process | where-object {$
>> | $_.WS -gt 100Mb
>> | sort-object WS -Descending
>> | select Name, WS
>> }

```

**Abb. 1.5** PSReadline vereinfacht das Editieren einer mehrzeiligen Eingabe

Das Modul umfasst fünf Cmdlets. Damit lassen sich allgemeine Einstellungen ändern und Tastaturshortcuts (neu) belegen. Die aktuelle Tastaturbelegung wird über das *Get-PSReadLineKeyHandler* ausgegeben.

### Beispiel

Der folgende Befehl belegt die Funktionstaste [F7] mit dem Start des Editors, der eine Profilskriptdatei lädt:

```
Set-PSReadlineKeyHandler -ScriptBlock { Notepad
$Profile.CurrentUserAllHosts } -Chord "F7"
```

*PSReadline* besitzt auch einen kleinen „Nachteil“. Die [F7]-Taste zeigt nicht mehr den Inhalt des Befehlsuffers der Konsole an, die Alternative ist die Eingabe von „h“ (dem Alias des *Get-History*-Cmdlets).

Oft möchte oder muss man die Farbeinstellungen ändern, da sich einzelne Vorder- und Hintergrundfarben mit der Hintergrundfarbe des Konsolenfensters „beißen“. Das ist natürlich ebenfalls kein Problem. Man muss dazu lediglich wissen, dass eine Farbe per *Set-PSReadlineOption*-Cmdlet pro Syntaxelement eingestellt werden muss und das Syntaxelement über den *TokenKind*-Parameter ausgewählt wird.

### Beispiel

Der folgende Befehl setzt die Hintergrundfarbe für Operatoren auf weiß:

```
Set-PSReadlineOption -BackgroundColor White -TokenKind Operator
```

Ein solcher Befehl wird in der Regel in der Profilskriptdatei untergebracht.

## 1.9 Hilfe

Die Hilfe zur PowerShell muss am Anfang einmal vom Microsoft-Server über das Internet abgerufen werden. Ansonsten steht nur eine Rumpfhilfe zur Verfügung, die lediglich aus einer Syntaxbeschreibung der Cmdlets besteht. Für das Abrufen der Hilfe gibt es zwei Cmdlets: *Update-Help* und *Save-Help*. Während *Update-Help* die Hilfe in das zuständige