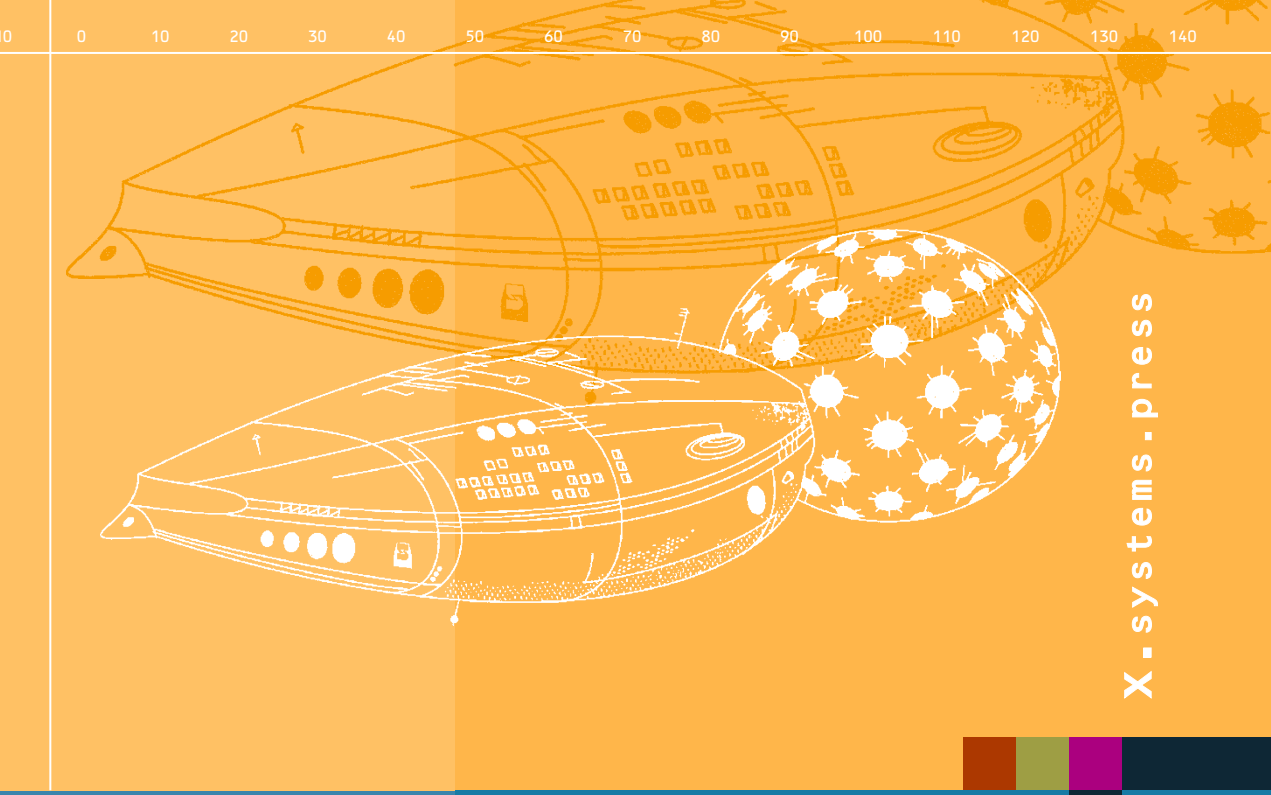

X.systems.press

Peter Monadjemi

Windows Server-Administration mit PowerShell 5.1

Eine kompakte und praxisorientierte Einführung

EBOOK INSIDE

Springer Vieweg

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

Springer Vieweg

Peter Monadjemi
Esslingen, Deutschland

ISSN 1611-8618 ISSN 2363-9059 (electronic)
X.systems.press
ISBN 978-3-658-17665-5 ISBN 978-3-658-17666-2 (eBook)
DOI 10.1007/978-3-658-17666-2

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

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 (vieles 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, 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

PowerShell für Kurzentschlossene

1

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.[1] 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.

[1] 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.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_1

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 *Desired State Configuration* 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	*Software Inventory Logging*. 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.

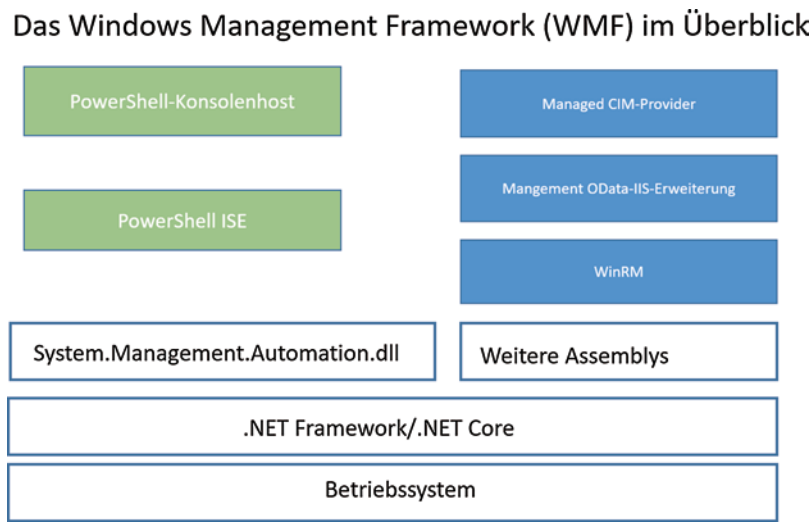

Abb. 1.1 Das WMF im Überblick

▶ **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-Assemblys 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 Assemblys

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 Typendefinitionen, 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 Betriff „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
```

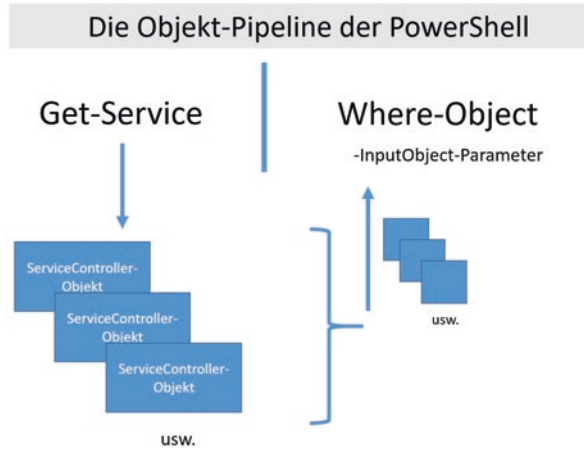

Abb. 1.2 Das Prinzip der Objekt-Pipeline – die Ausgabe eines Cmdlets wird in Gestalt von Objekten an den Parameter eines anderen Cmdlets gebunden

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.

▶ **Tipp** 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.

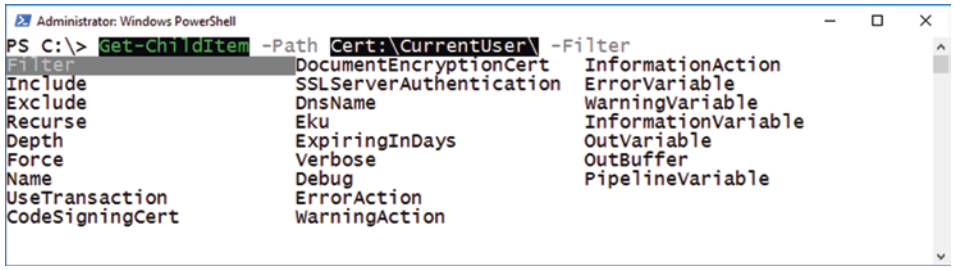

Abb. 1.3 Dank PSReadLine werden alle Parameter eines Cmdlets inklusive der dynamischen Parameter übersichtlich angezeigt

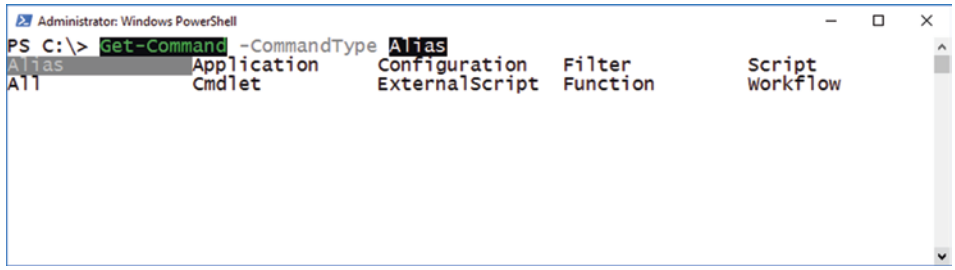

Abb. 1.4 Dank PSReadline werden auch die Parameterwerte eines Parameters übersichtlich angezeigt

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.[2] 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*.

▶ **Tipp** 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

[2] 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:\ProgramFiles\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.[3] 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.

[3] 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 PoweShell 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
```

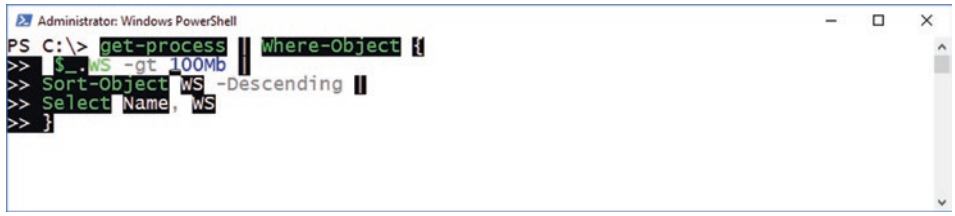

Abb. 1.5 PSReadline vereinfacht das Editieren einer mehrzeiligen Eingabe

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 Befehlspuffers 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

Unterverzeichnis im $PSHome-Verzeichnis ablegt, kann das Verzeichnis bei *Save-Help* frei festgelegt werden. Die Idee ist, dass ein Administrator die Hilfedatei in einem Verzeichnis ablegt und dieses gegebenenfalls freigibt, so dass seine Kollegen diese Dateien per *Update-Help* und dem *SourcePath*-Parameter aus diesem Verzeichnis auf ihren Computer übertragen.

- **Hinweis** Ist der Internetzugang nur über einen Proxy möglich, wird automatisch der im Internet Explorer konfigurierte Proxy verwendet.

- **Tipp** Sehr komfortabel wird die Hilfe in der PowerShell ISE zur Verfügung gestellt, indem Sie einfach die [F1]-Taste drücken, um zu einem Befehl, auf dem sich der Mauszeiger aktuell befindet, eine Hilfe zu erhalten.

1.10 Andere Plattformen

Seit August 2016 gibt es mit der PowerShell Core Edition eine Open Source-Version der PowerShell, die grundsätzlich auf jede Plattform portiert werden kann. Voraussetzung ist, dass sich .NET Core, eine Open Source-Version der .NET-Laufzeit, dort implementieren lässt. Seit WMF 5.1 gibt es die PowerShell in zwei Editionen: Desktop und Core. Die Core-Edition unterscheidet sich von der Desktop-Edition darin, dass sie keine plattformspezifischen Elemente enthält, ansonsten aber den vollständigen Funktionsumfang der PowerShell umfasst. Damit steht die PowerShell in der kommenden Version 6.0 auch für Linux und OSX zur Verfügung und natürlich für Windows. Sie zeichnet sich dort durch den Umstand aus, dass sie wie jede andere Anwendung auch im Programme-Verzeichnis abgelegt wird und damit auch von einem USB-Stick gestartet werden kann. Wie die PowerShell zum Beispiel unter *Ubuntu* eingesetzt wird, wird in Kap. 16 gezeigt.

Der Umgang mit Typen

2

Zusammenfassung

Typen und Objekte stehen bei der PowerShell im Mittelpunkt. In diesem Kapitel wird der Umgang mit Typen in seinen Facetten beschrieben. Es geht dabei auch um die öffentlichen und versteckten Members eines Objekts.

2.1 Alles ist ein Objekt

Bei der PowerShell ist alles ein Objekt. Alle Get-Commands geben Objekte zurück. Selbst eine harmlose Zeichenkette ist ein Objekt, die Members wie zum Beispiel eine *Length*-Eigenschaft besitzt. Auch Zahlen sind Objekte, da sie auf einfachen Typen basieren und entsprechend Members besitzen.

2.1.1 Objekte basieren auf Typen

Jedes Objekt basiert auf einer Typendefinition, kurz Typ. Jedes Objekt basiert damit auf einem Typ. Die meisten Typen sind in der PowerShell-Assembly *System.Management.Automation.dll* und in den zahlreichen Assembly-Dateien der .NET-Laufzeit definiert. Die meisten Typen sind Klassen (im Sinne von „Klassifizierung“), so dass die Begriffe Typ und Klasse bei der PowerShell wechselseitig austauschbar sind.

Alle Typen sind in Namespaces unterteilt. Hier ein Beispiel: Der Typ *CmdletInfo*, der bei der PowerShell allgemein ein Cmdlet repräsentiert, ist Teil des Namespaces *System.Management.Automation*. Dieser Name muss dem Typennamen offiziell vorangestellt werden. Die beiden Punkte im Namen deuten an, dass der Namespace hierarchisch organisiert ist. Es könnte daher auch Typen im Namespace *System.Management* geben.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_2

2.1.2 Den Typ eines Objekts herausfinden

Jedes Objekt basiert auf einem Typ und damit verbunden auf einer Typendefinition. Diese bestimmt zum Beispiel welche Members das Objekt anbietet. Es gibt mehrere Möglichkeiten, den Typ eines Objekts herauszufinden: Über das *Get-Member*-Cmdlet oder über die *GetType()*-Methode, die jedes Objekt als Member besitzt. Diese liefert ein *RuntimeType*-Objekt, dessen *FullName*-Eigenschaft den vollständigen und dessen Name-Eigenschaft den Kurznamen des Typs zurückgibt.

Beispiel

Der folgende Befehl gibt sowohl die kurze als auch die vollständige Typbezeichnung von dem Objekt aus, das bei der PowerShell einen Systemdienst repräsentiert:

```
(Get-Service -Name AudioSrv).GetType() | Select-Object Name, FullName
```

Der Name des Typs lautet „ServiceController“ beziehungsweise in der vollständigen Schreibweise, welche der Namespacename vorausgeht, „System.ServiceProcess.ServiceController“. Der Name „System.ServiceProcess“ ist der Namespacename, zu dem der Typ „ServiceController“ gehört.

2.2 Typen als Objekte

Was lässt sich mit einem Typennamen anfangen? Zwei Dinge: 1. Über einen Typennamen lassen sich Objekte per *New-Object*-Cmdlet anlegen, die auf diesem Typ basieren. 2. Aus dem Typennamen kann man auch selber ein Objekt machen, indem dieser in eckige Klammern gesetzt wird. Namen in eckigen Klammern stehen bei der PowerShell immer für Typenobjekte. Dahinter steckt ein Objekt vom Typ *RuntimeType*, über dessen Methoden alle Members des Typs abgefragt werden können.

Der folgende Befehl gibt über ein *RuntimeType*-Objekt die Namen aller Eigenschaften des Typs *System.ServiceProcess.ServiceController* aus:

```
[System.ServiceProcess.ServiceController].GetProperties().Name
```

Diese Namen erhält man auch über das *Get-Member*-Cmdlet. Wo ist der Unterschied beziehungsweise bietet eine der beiden Varianten Vorteile? Der Unterschied zwischen beiden Varianten ist subtil, aber wichtig. Das *Get-Member*-Cmdlet zeigt die Members auf der Grundlage vorhandener Objekte an. Bei der zweiten Variante ist kein Objekt im Spiel, hier wird der Typ direkt angesprochen. Vorteile bietet diese Variante nicht. Es ist lediglich eine andere Technik, bei der nur ein Typobjekt, aber keine Objekte im Spiel sind, die auf diesem Typ basieren.

2.3 Die Typen eines Arrays erhalten

Wendet man ein *Get-Member*-Cmdlet auf eine Array-Variable an, erhält man die Members der Objekte, die in dem Array enthalten sind, und nicht die Members des Arrays selber. Sollten diese Members aus irgendeinem Grund interessant sein, erhält man diese über jenes Objekt, dass die Array-Variable, wie jedes andere Objekt auch, über die *psbase*-Eigenschaft zur Verfügung stellt.

Beispiel

Das folgende Beispiel definiert ein Array und gibt die Members des Arrays aus:

```
$Zahlen = 10, 20, 30
$Zahlen.psbase | Get-Member
```

Die Eigenschaft *psbase* stellt das Basisobjekt zur Verfügung, auf dem ein anderes Objekt basiert.

Die zweite Alternative besteht darin, die Array-Variable dem *InputObject*-Parameter von *Get-Member* zu übergeben.

2.3.1 Auflisten der Konstruktoren einer Klasse

Eine Information, die das *Get-Member*-Cmdlet nicht über eine Klasse liefert, sind die Konstruktoren. Ein Konstruktor (engl. „constructor“) ist ein anderer Name für eine Methode der Klasse, die immer dann automatisch ausgeführt wird, wenn auf der Grundlage der Klasse ein neues Objekt, zum Beispiel per *New-Object*-Cmdlet, angelegt wird. Jede Klasse besitzt mindestens einen öffentlichen Konstruktor, damit sie überhaupt als Objekt angelegt werden kann. Gäbe es ihn nicht, könnte aus der Klasse kein Objekt gemacht werden. Neben diesem Default-Konstruktor kann es weitere Konstruktoren geben, denen auch Parameterwerte übergeben werden können.

▶ **Hinweis** Der Begriff „Konstruktor“ stammt aus dem Bereich der objektorientierten Programmierung. Mangels einer besseren Alternative wird er auch bei der PowerShell verwendet.

Ein Beispiel für einen Konstruktor, der Parameterwerte erwartet, ist der Konstruktor des Typs *PSCredential*. Dem Konstruktor werden zwei Parameterwerte übergeben: Ein Benutzername als String und ein Kennwort als Secure String. Für das Abfragen der Konstruktoren einer Klasse bietet das *RuntimeType*-Objekt die Methode *GetConstructors()*, die alle Konstruktoren als Objekte vom Typ *RuntimeConstructorInfo* liefert.

Der folgende Befehl gibt die Konstruktoren des Typs *PSCredential* zurück:

```
[PSCredential].GetConstructors()
```

Die Ausgabe ist noch etwas unübersichtlich, da neben den Parametern noch jede Menge anderer Details ausgegeben werden. Was bei einem Konstruktor in erster Linie interessant ist, sind die Namen und Datentypen der Parameter. Der Name des Konstruktors spielt keine Rolle, da er immer „.ctor“ lautet.

Beispiel

Der folgende Befehl gibt die Namen und Datentypen aller Konstruktoren des Typs *PSCredential* aus.

```
[PSCredential].GetConstructors().ForEach{ $_.GetParameters() | Select
Name, ParameterType }
```

Die Ausgabe macht deutlich, dass es beim Typ *PSCredential* noch einen zweiten Konstruktor gibt, dem ein einzelner Wert vom Typ *PSObject* übergeben wird. Durch die Ausgabe des Befehls erfährt man also, auf welche Weise ein Objekt vom Typ *PSCredential* per *New-Object*-Cmdlet angelegt werden kann.

Ganz optimal ist die Ausgabe des letzten Beispiels nicht, da die Namen und Typen der Parameter untereinander ausgegeben werden. Eine Zuordnung zu einem Konstruktor ist damit etwas schwierig. Besser wäre, die Parameter jedes Konstruktors zusammenzufassen.

Beispiel

Der folgende Befehl gibt erneut alle Konstruktoren des Typs *PSCredential* aus, dieses Mal aber etwas übersichtlicher:

```
[PSCredential].GetConstructors() | ForEach-Object -Begin { $i=0 } -
process {
  $i++;[PSCustomObject]@{Name="Constructor$i"; Parameter=
($_.GetParameters() | Select-Object @{n="Ctor"; e={"$($_.Name)
($($_.ParameterType))" }} | Select-Object -ExpandProperty Ctor) -join
","}
 }
```

Was für ein Aufwand. Und wer hat die Zeit und die „Brain-Power“, sich solche Konstruktionen auszudenken? Zum Glück gibt es seit der Version 5.0 eine deutlich einfachere Variante in Gestalt der statischen *New*-Methode des Typenobjekts. Wird diese Methode ohne das runde Klammerpaar angegeben, werden ebenfalls alle Konstruktorvarianten aufgelistet.

Beispiel

Der folgende Befehl gibt ebenfalls alle Konstruktoren der *PSCredential*-Klasse aus:

```
[PSCredendential]::New
```

Na also, so bleibt alles überschaubar. Falls Sie sich über das Verhalten wundern: Die PowerShell gibt generell bei Methodenmembers, die ohne das Klammerpaar angegeben werden, die Überladungsvarianten der Methode aus (zum Beispiel „[DateTime]::DaysInMonth“). Da auch *New* ein Methodenmember ist, funktioniert diese Technik auch mit den Konstruktoren der Klasse.

In den bisherigen Beispielen wurde der Typ *PSCredential* verwendet. Der Typ ist im Namespace *System.Management.Automation* enthalten. Der Grund dafür, dass der Namespacename bislang nicht angegeben wurde, ist, dass es sich bei *[PSCredential]* um einen Typenalias, also eine Abkürzung für eine Typenbezeichnung handelt (im Original „Type Accelerator“). Die PowerShell besitzt mehrere Dutzend solcher Typenaliase, durch die lange Typnamen abgekürzt werden. Wer möchte, kann sich auch eigene Typenalias anlegen.

2.4 Neue Objekte anlegen mit dem New-Object-Cmdlet

Das *New-Object*-Cmdlet legt ein Objekt mit dem über den *TypeName*-Parameter angelegten Typ an. Über den *ArgumentList*-Parameter werden Konstruktorwerte übergeben.

2.4.1 Neue Objekte anlegen über die statische Methode New

Seit der Version 5.0 besitzt das *RuntimeType*-Objekt, das einen Typen repräsentiert, eine statische Methode *New* für das Anlegen eines neuen Objekts, das auf dem Typ basiert.

Beispiel

Die folgende Befehlsfolge legt ein *PSCredential*-Objekt über die statische *New*-Methode des Typobjekts an.

```
$PwSec = "demo+123" | ConvertTo-SecureString –AsPlainText –Force
$Username = "pemo"
$Cred = [PSCredential]::new($Username, $PwSec)
```

Konstruktorargumente werden bei der *New*-Methode in runden Klammern übergeben.

2.4.2 Objekte mit Argumenten anlegen

Besitzt eine Klasse einen Konstruktor mit Parametern, können die Argumente (Parameterwerte) bei *New-Object* auf zwei Arten übergeben werden: Über den *ArgumentList*-Parameter oder indem die Argumente direkt auf den Namen der Klasse in runden Klammern folgen. Die erste Schreibweise ist die offizielle Schreibweise.

Beispiel

Der folgende Befehl legt per *New-Object*-Cmdlet ein Objekt vom Typ *PSCredential* an. Die Konstruktorwerte werden über den *ArgumentList*-Parameter übergeben.

```
$PwSec = "demo+123" | ConvertTo-SecureString –AsPlainText –Force
$Username = "pemo"
$Cred = New-Object –TypeName PSCredential –ArgumentList $Username, $PwSec
```

Wer nicht nur Administrator, sondern auch Software-Entwickler ist beziehungsweise in diesem Bereich bereits gearbeitet hat, ist es gewohnt, dass die Konstruktorwerte in runde Klammern gesetzt werden. Auch diese Schreibweise ist bei der PowerShell möglich.

Beispiel

Der folgende Befehl legt per *New-Object*-Cmdlet ein Objekt vom Typ *PSCredential* an. Die Konstruktorwerte werden dieses Mal in runden Klammern übergeben.

```
$PwSec = "demo+123" | ConvertTo-SecureString –AsPlainText –Force
$Username = "pemo"
$Cred = New-Object –TypeName PSCredential($Username, $PwSec)
```

2.4.3 Neue Objekte anlegen per [PSCustomObject]

Der Typenalias *[PSCustomObject]* erlaubt das Anlegen neuer Objekte vom Typ *System.Management.Automation.PSCustomObject*. Im Unterschied zu *New-Object* werden also immer Objekte desselben Typs angelegt. Namen und Werte der Eigenschaften werden über eine Hashtable angegeben.

2.5 Objektschreibweise

Bei Ad hoc-Abfragen in der Konsole werden die Members in der Regel über ihren Namen angesprochen. Bei Cmdlets wie *Select-Object*, *Sort-Object* oder *Where-Object* folgen die Namen der Eigenschaften des Objekts, das sich aktuell in der Pipeline befindet, einfach auf den Namen des Cmdlets. Ist keine Pipeline im Spiel, muss ein Member über den Punkt-Operator angesprochen werden.

Beispiel

Der folgende Befehl gibt über die *Count*-Eigenschaft die Anzahl der Dienste aus, die nicht automatisch starten:

```
(Get-Service | Where StartType –ne "Automatic").Count
```

Der unscheinbare Punkt trennt die Menge der Objekte, die von *Get-Service* geliefert werden, von der *Count*-Eigenschaft.

Beispiel

Der folgende Befehl setzt die Hintergrundfarbe für Fehlermeldungen für das Host-Fenster auf weiß.

```
$Host.PrivateData.ErrorBackgroundColor = "White"
```

Dass in diesem Befehl zwei Punkte vorkommen, kann einen PowerShell-Neuling zu Recht etwas irritieren. Der Hintergrund ist, dass *$Host* für ein Objekt steht und *PrivateData* eine Eigenschaft dieses Objekts ist. Doch da *PrivateData* wiederum für ein Objekt steht, darf ein weiterer Punkt folgen, um mit *ErrorBackgroundColor* eine Eigenschaft dieses Objekts anzusprechen.

2.6 Das versteckte Member PSObject

Jedes Objekt besitzt bei der PowerShell eine versteckte Eigenschaft mit dem Namen *PSObject*. Es liefert ein allgemeines Objekt vom Typ *PSObject*, über dessen Eigenschaften sich ebenfalls die Details eines Objekts, wie die Namen einer Members, abfragen lassen. Eine interessante Eigenschaft ist *TypeNames*, denn sie liefert die Namen der Typen, von denen sich der Typ, auf dem das Objekt basiert, ableitet (inklusive des aktuellen Typennamens).

2.7 Typen erweitern

Typen sind bei der PowerShell erweiterbar. Besitzt ein Typ eine bestimmte Eigenschaft nicht, kann diese im Rahmen der PowerShell-Session hinzugefügt werden. Warum sollte man das tun? Eine zwingende Anwendung gibt es dafür nicht. Diese Technik ist immer dann praktisch, wenn über eine Abfrage Objekte benötigt werden, die zusätzliche Eigenschaften anbieten, die Teil der Ausgabe sind.

- **Hinweis** Die PowerShell macht von dieser Erweiterbarkeit intensiven Gebrauch, indem an viele Typen der .NET-Laufzeit durch das PowerShell-Typensystem zusätzliche Eigenschaften angehängt werden. Damit sollen Abfragen zusätzliche Details liefern oder vorhandene Eigenschaften komfortabler abfragbar machen. Ein Beispiel ist der Typ *System.Diagnostics.Process*, der einen Prozess repräsentiert. Die Klasse *Process* der .NET-Laufzeit besitzt von Haus aus bereits 52 Eigenschaften. An diesen Typ werden 17 zusätzliche Members angehängt, unter anderem die Propertys *CPU*, Path und *WS*. Ohne diese angehängten Propertys wäre das Ergebnis der Abfrage nicht so praktisch auswertbar.

- **Tipp** Die nachträglich angehängten Members werden beim *Get-Member*-Cmdlet alleine ausgegeben, wenn der *View*-Parameter mit dem Wert „Extended“ verwendet wird.

Es gibt zwei Möglichkeiten, um einen Typ zu erweitern: über eine XML-Datei, die über das *Update-TypeData*-Cmdlet die Typeninformation der PowerShell aktualisiert, oder direkt über das *Update-TypeData*-Cmdlet ohne XML und mit den dafür vorgesehenen Parametern. Im Folgenden werden beide Methoden vorgestellt, die ein einfaches Ziel haben: an einen vorhandenen Typ ein weiteres Member für die Dauer der PowerShell-Sitzung anzuhängen.

Beispiel

Das folgende XML erweitert Objekte vom Typ *System.ServiceProcess.ServiceController* um eine Eigenschaft mit dem Namen „AnzahlAbhDienste“, die die Anzahl der von diesem Dienst abhängigen Dienste ausgibt.

```
$TypeData = @'
<Types>
    <Type>
        <Name>System.ServiceProcess.ServiceController</Name>
        <Members>
            <ScriptProperty>
                <Name>AnzahlAbhDienste</Name>
                <GetScriptBlock>
                 $this.DependentServices.Count
                </GetScriptBlock>
            </ScriptProperty>
        </Members>
    </Type>
</Types>
'@
```

Damit der Name der Variablen *$this* auch tatsächlich geschrieben und nicht mit einem (nicht existierenden) Wert ersetzt wird, wird der Here-String in Apostrophe und nicht in Anführungszeichen gesetzt.

Das XML wird in eine Datei mit der Erweiterung *.ps1xml* geschrieben. Der Name der Datei spielt keine Rolle, die Erweiterung muss *.ps1xml* sein.

```
$TypeData > ServiceTypeUpdate.ps1xml
```

Jetzt muss nur noch die Typeninformation der PowerShell erweitert werden, was das *Update-TypeData*-Cmdlet mit dem *AppendPath*-Parameter erledigt. Im Unterschied zum *PrependPath*-Parameter werden die neuen Typeninformationen erst angewendet, nachdem die „eingebauten“ Typeninformationen angewendet wurden.

```
Update-TypeData -AppendPath .\ServiceTypeUpdate.ps1xml
```

Damit steht die Eigenschaft „AnzahlAbhDienste“ bei jedem Objekt vom Typ *System.Diagnostics.ServiceController* im Rahmen der aktuellen PowerShell-Sitzung zur Verfügung und der folgende Befehl kann ausgeführt werden:

```
Get-Service | Select-Object Name, AnzahlAbhDienste
```

Bei der Ausgabe wird neben dem Namen eines Dienstes auch die Anzahl der von diesem Dienst abhängigen Systemdienste ausgegeben.

▶ **Hinweis** Eine selbst definierte Eigenschaft darf auch Leerzeichen enthalten. In diesem Fall muss der Name beim Ansprechen über Cmdlets wie *Select-Object* in Anführungsstriche gesetzt werden.

Die Erweiterbarkeit des Typensystems ist eine jener Eigenschaften der PowerShell, die sie besonders flexibel macht.

Die XML-Datei ist nur eine von zwei Möglichkeiten. Ein Member kann auch direkt über die Parameter des *Update-TypeData*-Cmdlets erweitert werden.

Beispiel

Der folgende Befehl erweitert per *Update-TypeData*-Cmdlet den Typ *System.ServiceProcess.ServiceController* um eine Eigenschaft „AbhDienste“, die die Namen der abhängigen Dienste enthält. Das, was beim Abrufen des Eigenschaftswertes passieren soll, wird über einen Scriptblock festgelegt, der dem *Value*-Parameter übergeben wird.

```
Update-TypeData -TypeName System.ServiceProcess.ServiceController -
MemberType ScriptProperty -MemberName "AbhDienste" Value {
($this.DependentServices | Select-Object -ExpandProperty Name) -join ","
} -Force

Get-Service | Select-Object Name, "AbhDienste"
```

Diese Variante, die erst seit der Version 3.0 zur Verfügung steht, ist natürlich deutlich weniger arbeitsaufwändig, als eine XML-Datei anlegen zu müssen. Auch hier wird der Typ, der erweitert werden soll, über die Spezialvariable *$this* angesprochen. Der *Force*-Parameter sorgt dafür, dass eine bereits vorhandene Memberdefinition überschrieben wird. Ein „Get-Service | Get-Member – View Extended“ zeigt die neue Eigenschaft *AbhDienste* an. Diese Eigenschaft gibt es aber nur im Rahmen der aktuellen PowerShell-Sitzung, sie wird nicht dauerhaft an die Typdefinition angehängt.

Dauert eine Abfrage eventuell länger, wenn Eigenschaften im Spiel sind, deren Wert berechnet werden muss? Eine Messung per *Measure-Command* ergibt, dass die Abfrage der abhängigen Dienste die Ausführungszeit um den Faktor 7 vergrößert. Doch da die Ausführungszeit immer noch im Bereich von Millisekunden liegt, dürfte sich das in der Praxis nicht auswirken.

Das Fazit dieses Abschnitts: Keine Scheu vor Typen. Bei PowerShell sind sie der Schlüssel für die Erweiterbarkeit von Objekten, so dass sich sehr flexible Abfragen zusammenstellen lassen.

2.8 Objekte erweitern über das Add-Member-Cmdlet

In diesem Abschnitt kommt es auf die Feinheiten an. Während das *Update-TypeData*-Cmdlet einen Typ um ein Member erweitert, erweitert das *Add-Member*-Cmdlet ein einzelnes Objekt um ein weiteres Member. Der Unterschied zum Anhängen der Members an den

Typ besteht ganz einfach darin, dass es die per *Add-Member* angehängten neuen Members nur bei diesem einen Objekt gibt. Da dies auch über das *Select-Object*-Cmdlet möglich ist, wird das *Add-Member*-Cmdlet nur für jene, in der Praxis eher selten vorkommende, Situationen benötigt, in denen ein bestimmter Typ von Member hinzugefügt werden soll.

Beispiel

Der folgende Befehl hängt bei allen Objekten vom Typ *FileInfo*, die zum Beispiel vom *dir*-Kommando geliefert werden, ein ScriptProperty-Member mit dem Namen „Comment“ an, in die die Kommentarzeile der Skriptdatei gehängt wird. Damit es einfach bleibt, werden immer die ersten drei Zeilen der Ps1-Datei verwendet, unabhängig davon, ob diese einen Kommentar enthalten oder nicht.

```
<#
 .Synopsis
 Objekt-Erweiterung per Add-Member
#>

$SB = {
  (Get-Content -Path $this.FullName -TotalCount 3) -join "`n"
}

dir -Path *.ps1 | ForEach-Object {
    $_ | Add-Member -MemberType ScriptProperty -Name Comment -Value $SB -
PassThru
} | Format-List Name, Comment
```

Für jede Ps1-Datei werden untereinander ihr Name und der Inhalt der ersten drei Zeilen ausgegeben. Wie allgemein bei einer Typenerweiterung muss auch hier der Typ über die Spezialvariable *$this* angesprochen werden.

Wie wäre es mit einer kleinen Übung? Wie kann die Eigenschaft „Comment“ über das *Update-TypeData*-Cmdlet dauerhaft hinzugefügt werden? Tipp: Der Typ, auf den bei der .NET-Laufzeit eine Datei basiert, heißt „System.IO.FileInfo“.

2.9 Typen definieren über externen Code

Bis zur Version 4.0 der PowerShell war es nur per externem Programmcode, der in einer Programmiersprache wie C# oder Visual Basic geschrieben wurde, möglich, eigene Typen zu definieren. Der Programmcode wird über das universelle *Add-Type*-Cmdlet kompiliert und damit wird die Typendefinition Teil der aktuellen PowerShell-Sitzung. Im Anhang wird diese Technik vorgestellt.

Seit der Version 5.0 besteht die Möglichkeit, über den *class*-Befehl Typen zu definieren, aus denen sich per *New-Object*-Cmdlet anschließend Objekte machen lassen. Dies ist eine deutliche Vereinfachung gegenüber der Notwendigkeit, den Typ per Programmcode definieren zu müssen. Da der *class*-Befehl in diesem Buch in Kap. 3 vorgestellt wird, bleibt es an dieser Stelle bei einem einfachen Beispiel.

Beispiel

Das folgende Beispiel definiert über den *class*-Befehl eine Klasse (also einen Typen) mit dem Namen „Ps1File“.

```
<#
 .Synopsis
 Typendefinition über den class-Befehl
#>

class Ps1File
{
  [string]$Ps1Path
  [string]$Ps1Name

  Ps1File([string]$Ps1Path)
  {
    $this.Ps1Path = $Ps1Path
    $this.Ps1Name = Split-Path -Path $Ps1Path -Leaf
  }

  [string]Comment()
  {
      return (Get-Content -Path $this.Ps1Path -TotalCount 3) -join "`n"
  }
}
```

Auch bei diesem Beispiel dürfte sich bei den meisten Lesern als Kommentar vermutlich ein „Und nun?“ aufdrängen. Was kann man mit dem neuen Typen denn Schönes (und vor allem Nützliches) anfangen? Nicht allzu viel, das ist klar. Ein Typ ist immer dazu da, mit seiner Definition neue Objekte anlegen zu können.

Beispiel

Das erledigt der folgende Befehl, der alle Ps1-Dateien im aktuellen Verzeichnis als Ps1File-Objekte ausgibt.

```
dir -Path *.ps1 | ForEach-Object {
    $Ps1File = New-Object -TypeName Ps1File -ArgumentList $_.FullName
     $Ps1File | Select -Property Ps1Name, @{n="Comment";e={$_.Comment()
}}
}
```

Der, zugegeben kleine, Vorteil dieser Varianten besteht darin, dass keine Objekte vom Typ *PSCustomObject*, sondern eines speziellen Typs, in diesem Fall Ps1File, ausgegeben werden. Über das *Update-FormatData*-Cmdlet könnte diesem Typ zum Beispiel eine eigene Formatierung zugeordnet werden. Oder Objekte dieses Typs könnten auf eine bestimmte Art und Weise serialisiert (also in ein Textformat konvertiert) werden, damit sie sich im Rahmen einer Remoting-Verbindung effektiver übertragen lassen. Diese beiden Beispiele machen deutlich, dass benutzerdefinierte Typen nur in spezielleren Konstellationen ihre Vorteile ausspielen.

2.9.1 Zusammenfassung

Bei der PowerShell stehen Objekte im Mittelpunkt. Jedes Get-Command gibt seine Rückgabe in Gestalt von Objekten zurück. Objekte basieren auf Typen. Der Typ legt fest, welche Members ein Objekt besitzen. Die meisten Typen stammen aus der .NET-Laufzeit. Das ist daran zu erkennen, dass der Namespace mit „System“ beginnt. Prozesse basieren auf dem Typ *System.Diagnostics.Process*, Dienste auf dem Typ *System.SystemProcess.ServiceController*, Dateien auf dem Typ *System.IO.FileInfo*. AD-Benutzerkonten auf dem Typ *Microsoft.ActiveDirectory.Management.ADUser*. Dieser Typ ist in einer Assembly(-Datei) definiert, die Teil des ActiveDirectory-Moduls ist.

Klassen definieren mit dem *class*-Befehl

3

Zusammenfassung

Seit der Version 5.0 besteht über den *class*-Befehl der PowerShell die Möglichkeit, in einem Skript Klassen zu definieren, aus denen Objekte gemacht werden können (aber nicht müssen). In erster Linie wurde diese Technik eingeführt, damit sich DSC-Ressourcen mit weniger Aufwand erstellen lassen. Klassen können aber auch für andere Zwecke verwendet werden. Sie sind allgemein ein praktisches Programmierelement, mit dem sich beliebige Datenstrukturen abbilden lassen. Als solche sind sie fester Bestandteil nahezu aller Programmier- und Skriptsprachen. Alleine aus diesem Grund ist es wichtig, dass es sie auch bei der PowerShell gibt.

Für Leser, die nicht nur Administrator, sondern auch Software-Entwickler sind oder sich in diesem Bereich auskennen: Das Klassenkonzept wurde bei der PowerShell absichtlich einfach gehalten und orientiert sich eher an Python als an C#. In kommenden Versionen soll es zwar kleinere Verbesserungen geben, das Ziel ist es aber nicht, alle Möglichkeiten nachzubauen, die es bei richtigen Programmiersprachen wie C# oder Java gibt.

3.1 Einleitung

Wir beginnen mit einer Begriffsdefinition. Eine Klasse ist eine andere Bezeichnung für das Wort Typ. Ein Typ definiert, welche Members ein Objekt besitzt. Jedes Objekt basiert daher auf einem Typen. Ein anderer Begriff für Typ ist Typendefinition. Mit einer Klasse wird daher eine Schablone definiert, die alle Members umfasst. Wird aus der Klasse ein

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_3

Tab. 3.1 Neue Befehlswörter für die Definition von Klassen

Befehl	Bedeutung
Class	Definiert eine Klasse.
Enum	Definiert eine Enumeration.
hidden	Macht ein Member unsichtbar für *Get-Member* (sofern dessen *Force*-Parameter nicht verwendet wird).
Static	Deklariert ein Member als statisch, so dass es per ::-Operator direkt über den Namen der Klasse angesprochen wird.

Objekt gemacht, zum Beispiel per *New-Object*-Cmdlet, enthält das Objekt alle Members, die in der Klasse definiert sind.

Begriffe wie Klassen und Typen sind für Menschen, die sich in ihrem bisherigen Leben wenig bis gar nicht dem Thema Programmieren von Software beschäftigt haben, am Anfang schwer zu fassen, da sie allgemeinen Begriffen aus dem Alltag entsprechen und sich daher nur schwer auf etwas ganz anderes übertragen lassen. Denken Sie bei Klasse nicht an die Schulklasse und bei Typ nicht an die umgangssprachliche Umschreibung für einen Mitmenschen männlichen Geschlechts. Sehen Sie den Begriff „Klasse“ unter dem Aspekt des „Klassifizierens“, also des Einteilens von Gegenständen in bestimmte Schubladen. Auch der Begriff Typ für „Typisieren“, also Einteilen, muss unter diesem Aspekt betrachtet werden.

► **Tipp** Das Thema Umgang mit Klassen wird in der PowerShell-Hilfe unter dem Thema „about_Classes“ ausführlich und mit Beispielen beschrieben. Das Hilfethema bezieht sich in erster Linie auf das Definieren von DSC-Ressourcen mit Hilfe von Klassen. Das Thema Definieren von Typen wird lediglich am Ende des Hilfetextes kurz behandelt.

3.1.1 Befehlswörter für die Definition von Klassen

Die Definition von Klassen wird durch neue Befehlswörter unterstützt, die in Tab. 3.1 zusammengefasst sind.

3.2 Klassen definieren

Eine Klasse wird mit dem *class*-Befehl definiert, auf den der Name der Klasse folgt. Die Klassendefinition wird, wie bei der PowerShell üblich, in geschweifte Klammern gesetzt, wenngleich damit ausnahmsweise kein Scriptblock definiert wird.

Beispiel

Das folgende Beispiel definiert eine Klasse mit dem Namen „UserData“.

```
<#
 .Synopsis
 Definition einer Klasse
#>

class UserData

{

}
```

Was lässt sich mit der neuen Klasse anfangen? Die Eingabe von „Userdata“ in die Konsole, nachdem die Klassendefinition einmal ausgeführt wurde, führt zu einer Fehlermeldung. Name nicht bekannt. Kein Wunder, denn die neue Klasse ist ein Typ und der Name muss daher in eckige Klammern gesetzt werden. Ein „[Userdata]“ führt zu einer Ausgabe des *RuntimeType*-Objekts, das diesen Typen repräsentiert. Die vollständige Typenbezeichnung wird über ein „[UserData].FullName“ ausgegeben. *FullName* ist eine Eigenschaft des *RuntimeType*-Objekts, die die vollständige Typenbezeichnung enthält.

Die eigentliche Daseinsberechtigung einer Klasse besteht darin, daraus Objekte (in diesem Zusammenhang auch „Instanzen“ genannt) zu machen. Zum Beispiel über die statische *New*-Methode des Typenobjekts:

```
[UserData]::new()
```

Das Ergebnis ist ein Objekt, das noch keine Eigenschaften besitzt. Das soll im nächsten Abschnitt geändert werden.

3.3 Hinzufügen von Eigenschaften

Eine Eigenschaft wird durch eine reguläre Variablendefinition hinzugefügt.

Beispiel

Das folgende Beispiel fügt zur Klasse *UserData* eine Eigenschaft „UserName“ hinzu.

```
<#
 .Synopsis
 Definition einer Klasse mit Eigenschaft
#>

class UserData
{
  $UserName
}
```

Beispiel

Der folgende Befehl gibt alle Members des neuen Typs mit ihren Namen, dem Membertyp und dem Typ des Rückgabewertes aus, wenn es sich um ein Methoden-Member handelt.

```
[UserData].GetMembers() | Select Name, MemberType, ReturnType
```

Beispiel

Der folgende Befehl gibt nur die Property-Members mit ihrem Namen und ihrem Typen aus.

```
[UserData].GetProperties() | Select Name, PropertyType
```

Die Ausgabe macht drei Eigenschaften des Members *UserName* deutlich:

1. Es ist eine Property.
2. Die Property kann gelesen und geschrieben werden, da es einen Get-Accessor (*get_UserName*) und einen Set-Accessor (*set_Username*) gibt.
3. Die Eigenschaft ist vom Typ *Object.*

▶ **Hinweis** Eine ReadOnly-Eigenschaft gibt es bei der PowerShell 5.1 nicht. Es ist lediglich möglich und sinnvoll, jeder Eigenschaft einen expliziten Typen zu geben.

Beispiel

Das folgende Beispiel definiert die Eigenschaft *UserName* als Eigenschaft vom Typ String.

```
<#
 .Synopsis
 Definition einer Klasse mit einer typisierten Eigenschaft
#>

class UserData
{
  [String]$UserName
}
```

▶ **Hinweis** Streng genommen ist eine Variable innerhalb einer Klassendefinition ein Feld (engl. „field“) und keine Eigenschaft, da es keine separaten get- und set-Zugriffsmethoden für das Abrufen und Setzen des Eigenschaftswertes gibt. Doch da die Variable intern als Property geführt und per *GetProperty*-Methode des *RuntimeType*-Objekts geholt wird, sind Variablen innerhalb einer Klassendefinition offiziell Eigenschaften. Soll eine Eigenschaft einen berechneten Wert zurückgeben, muss sie als Methode implementiert werden.

3.4 Aus Klassen Objekte machen

Eine Klasse ist nur eine Definition. Wenn sie für irgendetwas Praktisches verwendet werden soll, muss aus ihr ein Objekt gemacht werden. Das übernimmt entweder die statische *New*-Methode oder das *New-Object*-Cmdlet.

Beispiel

Der folgende Befehl legt mit der Klasse *UserData* ein Objekt an und weist die Referenz auf das Objekt, das Sie sich als einen kleinen Bereich im Arbeitsspeicher vorstellen müssen, in dem die Werte der Property-Members abgelegt sind, einer Variablen mit dem Namen „U1".

```
$U1 = New-Object –TypeName UserData
```

Über die Variable *U1* wird der Wert der Eigenschaft *UserName* mit dem Punkt-Operator angesprochen.

```
$U1.UserName = "Erwin L."
```

3.5 Statische Member

Ein statisches Member ist ein Member, das direkt über die Klasse mit Hilfe des Operators zum Aufruf statischer Members :: angesprochen wird. Der Memberdefinition muss dazu das Schlüsselwort *Static* vorangestellt werden. Wenn ein Member statisch gemacht wird, geht es im Allgemeinen um Methoden-Members.

Beispiel

Das folgende Beispiel definiert in der Klasse *UserData* eine statische Eigenschaft mit dem Namen „Index".

```
<#
 .Synopsis
 Definition einer Klasse mit einem statischen Member
#>

class UserData
{
  [String]$UserName
  Static[Int]$Index
}
```

Beispiel

Das folgende Beispiel weist der statischen Eigenschaft *Index* einen Wert zu.

```
[UserData]::Index = 1
```

Möchte man über den Namen einer Eigenschaft feststellen, ob diese statisch ist oder nicht, ist das nicht möglich, da es bei der .NET-Laufzeit intern keine Eigenschaften gibt, sondern nur einen *get*- und einen *set*-Accessor. Die Abfrage muss daher über die Abfrage der Methoden erfolgen.

Beispiel

Der folgende Befehl bestätigt, dass eine *Index*-Eigenschaft statisch ist, indem die *get_Index*- und *set_Index*-Methode zurückgegeben wird.

```
[UserData].GetMethods() | Where IsStatic | Select Name, IsStatic
```

3.6 Enumerationen

Eine Enumeration ist ein anderer Name für eine Liste von Konstanten, die unter einem gemeinsamen Namen zusammengefasst werden. Damit besteht die Möglichkeit, einer Eigenschaft einen Datentyp zu verleihen, der nur einer der Konstantenwerte sein kann, die Teil der Enumeration sind. Wird ein anderer Wert zugewiesen, ist eine Fehlermeldung die Folge.

▶ **Hinweis** Eine Einschränkung ist, dass Enumerationswerte nur Zahlen vom Typ Int32 sind.

Eine Enumeration wird über den *enum*-Befehl definiert und bei einer Variablen anstelle des Datentyps eingesetzt.

Beispiel

Das folgende Beispiel definiert eine Enumeration mit dem Namen „UserType“.

```
<#
 .Synopsis
 Definition einer Enumeration
#>

enum UserType
{
  Domain
  Local
}
```

Die neue Enumeration kann zum Beispiel anstelle eines regulären Typs bei einer Eigenschaft oder als Rückgabewert einer Methode verwendet werden.

Beispiel

Das folgende Beispiel fügt in der Klasse *UserData* eine Eigenschaft vom Typ *UserType* mit dem Namen „UserType“ ein. Der Umstand, dass der Name der Eigenschaft dem Namen der Enumeration entspricht, ist nicht ganz optimal, grundsätzlich aber kein Problem.

```
<#
 .Synopsis
 Definition einer Klasse mit einem Enumerations-Member
#>

enum UserType
{
  Domain
  Local
}

class UserData
{
  [String]$UserName
  Static[Int]$Index
  [UserType]$UserType
}
```

Die Enumerationskonstante kann auf zwei Arten zugewiesen werden: Formal über den Typen der Enumeration und der Auswahl der Konstanten. Weniger formal, indem der Name der Konstanten in Anführungszeichen gesetzt wird.

Beispiel

Der folgende Befehl führt die formale Zuweisung eines Wertes der Enumeration durch.

```
$u1 = [UserData]::new()
$u1.UserType = [UserType]::Domain
```

Beispiel

Der folgende Befehl führt die kurze Form der Zuweisung eines Wertes der Enumeration durch.

```
$u1 = [UserData]::new()
$u1.UserType = "Domain"
```

Dass auch hinter einer Enumeration ein eigener Typ steckt, macht der folgende Befehl deutlich, der die statischen Members des Typs und damit die Konstanten ausgibt.

```
[UserTyp] | Get-Member -Membertype Property –Static
```

3.7 Aus Klassen werden Objekte

Eine Klassendefinition alleine ist relativ nutzlos, denn es lassen sich lediglich die statischen Members nutzen, sofern diese vorhanden sind. Damit eine Klasse einen Zweck erfüllt, muss aus ihr ein Objekt (in diesem Zusammenhang auch Instanz genannt) gemacht werden. Das ist bei PowerShell auf zwei Arten möglich: Über das vertraute *New-Object*-Cmdlet und über die statische *New*-Methode, die ab der Version 5.0 jedes Typenobjekt besitzt.

Beispiel

Der folgende Befehl legt ein Objekt der Klasse *UserData* per *New-Object* an.

```
New-Object -TypeName UserData
```

Beispiel

Der folgende Befehl legt ein Objekt per *New*-Methode der Klasse an.

```
[UserData]::New()
```

Auf welche Weise das Objekt angelegt wird, spielt keine Rolle. Soll ein Skript auch unteren älteren PowerShell-Versionen ausführbar sein, muss generell *New-Object* verwendet werden. In dieser Umgebung gibt es natürlich auch keinen *class*-Befehl.

3.8 Klassen mit einem Konstruktor

Ein Konstruktor ist ein Begriff, der aus der Welt der objektorientierten Programmierung stammt. Er steht für eine Methode, die den Namen der Klasse trägt, und die automatisch immer dann ausgeführt wird, wenn aus der Klasse ein Objekt gemacht wird. Bei der PowerShell ist dies immer dann der Fall, wenn entweder das *New-Object*-Cmdlet mit dem Namen der Klasse als Typenbezeichnung ausgeführt oder die statische *New*-Methode aufgerufen wird.

Beispiel

Das folgende Beispiel erweitert die Klasse *UserData* um einen Konstruktor. Er erhöht die statische Eigenschaft *Index* um eins. Der Wert dieser Eigenschaft wird der nicht statischen Eigenschaft *Id* zugewiesen.

```
<#
 .Synopsis
 Definition einer Klasse mit einem Konstruktor
#>

enum UserType
{
  Domain
  Local
}

class UserData
{
  [String]$UserName
  Static[Int]$Index
  [UserType]$UserType
  [Int]$Id
  UserData()
  {
    [UserData]::Index++
    $this.Id = [UserData]::Index
  }
}
```

Ganz nebenbei zeigt das Beispiel, wie innerhalb der Klassendefinition eine statische Eigenschaft angesprochen wird. Dabei wird der Name der Klasse in eckige Klammern gesetzt und die statische Eigenschaft per :: angesprochen.

3.8.1 Konstruktor mit Parameter

Wie eine Methode kann ein Konstruktor einen oder mehrere Parameter besitzen. Auch wenn dies in der Praxis nur selten eine Rolle spielen dürfte, zieht die PowerShell auch in diesem Punkt mit anderen objektorientierten Skriptsprachen gleich.

Beispiel

Das folgende Beispiel definiert eine Klasse mit einem Konstruktor mit einem *Int*-Parameter, so dass beim Anlegen eines Objekts auch ein Wert, der sich in einen Integer konvertieren lässt, zur Verfügung gestellt werden muss.

```
<#
 .Synopsis
 PowerShell-Klasse mit Konstruktor und einem Parameter
#>

class UserData
{
  [int]$Id

  UserData([Int]$UserId)
  {
    $this.Id = $UserId
  }
}
```

Beispiel

Der folgende Aufruf verwendet die statische *New*-Methode für das Anlegen des Objekts.

```
$user1 = [UserData]::new(10)
```

Beispiel

Der folgende Aufruf verwendet das *New-Object*-Cmdlet und den *ArgumentList*-Parameter.

```
$user2 = New-Object -TypeName UserData -ArgumentList 20
```

3.8.2 Versteckte Members

Über das Schlüsselwort *hidden* wird ein Member als versteckt deklariert. Ein verstecktes Member kann angesprochen werden, erscheint aber nicht in der Auswahlliste. Versteckte Members werden bei *Get-Member* nur per *Force*-Parameter ausgegeben.

Beispiel

Das nächste Beispiel enthält eine etwas praxisnähere Anwendung für eine eigene Klasse. Es definiert eine Klasse mit dem Namen „PSCredentialEx“, deren Konstruktor zwei String-Werte erwartet. Damit kann ein *PSCredential*-Objekt auch mit einem Klartext-Kennwort angelegt werden. Über die Methode *GetCredential()* ist das resultierende *PSCredential*-Objekt abrufbar. Damit die beiden Eigenschaften *Username* und *Credential* „offiziell“ nicht direkt mit Werten belegt werden können, werden sie per *hidden* „versteckt“. Eine Zuweisung ist zwar trotzdem möglich, aber die Wahrscheinlichkeit, dass die Eigenschaften von einem über die wahre Bedeutung der Klasse uninformierten User zweckentfremdet werden, wird etwas reduziert.

```
<#
 .Synopsis
 Eine PowerShell-Klasse mit versteckten Members
#>

class PSCredentialEx
{
  hidden[String]$UserName
  hidden[PSCredential]$Credential

  PSCredentialEx([String]$Username, [String]$Password)
  {
   $this.UserName = $Username
   $PwSec = $Password | ConvertTo-SecureString -AsPlainText –Force
   $this.Credential = [PSCredential]::new($Username, $PwSec)
  }

  [PSCredential]GetCredential()
  {
   return $this.Credential
  }
}
```

Beispiel

Das folgende Beispiel macht aus der Klasse *PSCredentialEx* ein Objekt. Die *GetCredential()*-Methode liefert das aus Benutzername und Kennwort zusammengebaute *PSCredential*-Objekt.

```
$Username = "Administrator"
$Password = "demo+123"
$Cred = New-Object -TypeName PSCredentialEx -ArgumentList $Username,
$Password
$Cred.GetCredential()
```

3.8.3 Überladene Konstruktoren

Eine überladene Methode ist eine Methode, deren Namen in einer Klassendefinition mehrfach vorkommt. Da die einzelnen Methoden unterschieden werden können, unterscheiden sie sich durch die Anzahl und/oder Typen der Parameter. Neben Methoden kann auch der Konstruktor überladen werden.

Beispiel

Das folgende Beispiel definiert eine Klasse `UserData` mit zwei Konstruktoren. Während der erste Konstruktor ohne Parameter definiert wird, verwendet der zweite Konstruktor einen *Int*-Parameter.

```
<#
 .Synopsis
 PowerShell-Klasse mit einem überladenden Konstruktor - einmal mit und
einmal ohne Parameter
#>

class UserData
{
  [int]$Id

  UserData ()
  {
    $this.Id = -1
  }

  UserData ([Int]$UserId)
  {
    $this.Id = $UserId
  }
}
```

Damit kann aus der Klasse *UserData* sowohl ohne Konstruktor als auch mit einem Parameterwert für den Konstruktor ein Objekt gemacht werden.

Beispiel

Der folgende Aufruf legt ein Objekt mit dem Standardkonstruktor an.

```
[UserData]:New()
```

Beispiel

Der folgende Aufruf legt ein Objekt mit dem Konstruktor an, der einen Parameterwert erwartet.

```
[UserData]:New(1)
```

▶ **Tipp** Eine Liste aller Konstruktorüberladungen einer Klasse liefert die Eingabe der statischen *New*-Methode des Typenobjekts ohne ein Paar runder Klammern.

Beispiel

Der folgende Befehl gibt alle Konstruktoren der Klasse *UserData* aus.

```
[UserData]::new
```

3.8.4 Klassen mit einem statischen Konstruktor

Eine Klasse kann auch einen statischen Konstruktor besitzen. Dieser wird mit dem Befehlswort *static* deklariert und einmalig aufgerufen, wenn die Klasse das erste Mal verwendet wird.

Beispiel

Das folgende Beispiel definiert eine Klasse mit einem statischen Konstruktor, der eine statische Eigenschaft mit einem Wert vorbelegt. Der Zugriff auf die nicht-statischen Eigenschaften und Methoden ist nicht möglich.

```
<#
 .Synopsis
 PowerShell-Klasse mit einem statischen Konstruktor
#>

class UserData
{
  static[int]$InstanceCount
  static UserData()
  {
    [UserData]::InstanceCount++
  }
}
```

Mit dem ersten Verwenden der Klasse *UserData* erhält die Eigenschaft *InstanceCount* den Wert 1. Anders als bei einem nicht-statischen Konstruktor wird der Wert mit jedem weiteren Verwenden der Klasse nicht mehr erhöht, da ein statischer Konstruktor nur einmal ausgeführt wird.

3.9 Methoden definieren

Eine Methode ist bekanntlich eine weitere Sorte von Member, die bei der PowerShell für eine Function steht, die Teil der Klassendefinition ist.

Beispiel

Das folgende Beispiel definiert in der Klasse *UserData* eine Methode *GetDaysSinceLastLogon*.

```
<#
 .Synopsis
 PowerShell-Klasse mit einer Methoden-Definition
#>

enum UserType
{
  Domain
  Local
}

class UserData
{
  [UserType]$UserType

  UserData()
  {
      $this.UserType = "Domain"
  }

  UserData([UserType]$UserType)
  {
      $this.UserType = $UserType
  }

  [int]GetDaysSinceLastLogon()
  {
    return 42
  }

}
```

Mit einem Methoden-Member gehen drei Besonderheiten einher, die sie von einer Function unterscheiden:

1. Gibt eine Methode einen Wert zurück, muss dieser mit dem *return*-Befehl zurückgegeben werden.
2. Wenn eine Methode einen Wert zurückgibt, muss der Datentyp des Rückgabewertes dem Namen der Methode vorangehen.
3. Greift die Methodenimplementierung auf eine Eigenschaft der Klasse zu, muss diese über *$this* angesprochen werden.

Einen zunächst ungewohnten Unterschied gibt es, wenn innerhalb einer Methodendefinition eine Eigenschaft der Klasse angesprochen werden soll. Die Variable muss über die Variable *$this* angesprochen werden, die im Rahmen einer Klassendefinition für das Objekt steht, in dem der Befehl später ausgeführt wird.

Der Umstand, dass die Methode nur einen Dummy-Wert zurückliefert, ist dem Umstand geschuldet, dass für das Testen kein Active Directory zur Verfügung steht und das *Get-ADUser*-Cmdlet daher nicht ausgeführt werden kann. Dies ist ein klassischer Fall für Mock-Commands. In Kap. 6 wird gezeigt, wie sich mit Hilfe des *Pester*-Moduls auch dann Tests für den Aufruf von Cmdlets schreiben lassen, die bei der Ausführung des Tests aus irgendeinem Grund nicht zur Verfügung stehen.

3.9.1 Methoden überladen

Das Prinzip der Überladung wurde bereits beim Konstruktor beschrieben. Auch Methoden können überladen werden. Der kleine Vorteil der Überladung ist, dass wenn eine Methode mit mehreren Parametern aufrufbar sein soll, nicht alle Parameter in eine Methodendefinition gepackt werden müssen. Ein weiterer Vorteil ist, dass eine Methode der Basisklasse durch eine eigene Implementierung ersetzt werden kann, die denselben Namen trägt wie die Methode der Basisklasse.

Beispiel

Das folgende Beispiel definiert in der Klasse *UserData* erneut eine Methode „GetDaysSinceLastLogon“, der dieses Mal ein *DateTime*-Wert übergeben wird. Damit kann die Methode einmal ohne und einmal mit einen (DateTime-) Wert aufgerufen werden.

```
<#
 .Synopsis
 PowerShell-Klasse mit einer überladenen Methoden-Definition
#>

enum UserType
{
  Domain
  Local
}

class UserData
{
  [UserType]$UserType

  UserData()
  {
    $this.UserType = "Domain"
  }

  UserData([UserType]$UserType)
  {
    $this.UserType = $UserType
  }

  [int]GetDaysSinceLastLogon()
  {
    return 42
  }

  [int]GetDaysSinceLastLogon([DateTime]$Datum)
  {
    return ((Get-Date)-$Datum).Days
  }

}
```

Die Frage, welche der beiden gleichnamigen Methoden aufgerufen wird, beantwortet sich durch den Parameterwert. Wird einer übergeben, wird die Variante mit dem

Parameter aufgerufen, ansonsten jene, die ohne Parameter definiert wurde. Bei der Übergabe eines Datumswertes gilt erneut zu beachten, dass das Datum im Format „Monat-Tag-Jahr“ übergeben werden muss, da die Zeichenfolge ansonsten nicht in einen *DateTime*-Wert konvertiert werden kann.

Beispiel

Der folgende Befehl ruft die Methode *GetDaysSinceLastLogon()* mit einem Parameter auf.

```
$user1 = [UserData]::new()
$use1.GetDaysSinceLastLogon("01/01/2017")
```

3.9.2 Statische Methoden

Auch Methoden können wie Eigenschaften statisch sein. Damit existiert bei einer Klasse eine Methode, die direkt über die Klasse aufgerufen wird. Statische Methoden können nicht auf Eigenschaften oder Methoden der Klasse zugreifen, die nicht statisch sind. Ein Grund für eine statische Methode ist, dass damit eine Aufrufvariante zur Verfügung steht, die ein Objekt liefert, bei dem einzelne Eigenschaften bereits mit bestimmten Werten vorbelegt sind.

Beispiel

Das folgende Beispiel definiert in der Klasse *UserData* eine statische Methode „GetNewUserData()“, die ein *UserData*-Objekt zurückgibt. Es ist bemerkenswert, dass die Definition der statischen Methode bereits etwas anspruchsvoller ist, da neben dem Befehlswort *static* der eigene Typ als Rückgabewert festgelegt wird.

```
<#
 .Synopsis
 PowerShell-Klasse mit einer statischen Methode
#>


class UserData
{
  [Int]$UserId

  UserData([Int]$UserId)
  {
    $this.UserId = $UserId
  }

  static[UserData]GetNewUserData()
  {
    return [UserData]::new(-1)
  }
}
```

Beispiel

Der folgende Befehl gibt über die statische Methode *GetNewUserData()* ein Objekt vom Typ *UserData* zurück, bei dem die *UserId*-Eigenschaft mit einem Wert vorbelegt ist.

```
[UserData]::GetNewUserData()
```

3.10 Klassen ableiten (Vererbung)

Bei der PowerShell kann eine Klasse von einer bereits vorhandenen Klasse abgeleitet werden. Sie übernimmt damit alle Members dieser Klasse und kann diese bei Bedarf durch eine eigene Implementierung ersetzen (überschreiben). Diese Technik wird auch als Vererbung bezeichnet, da die neue Klasse alle Members der anderen Klasse „erbt". Die Klasse, von der abgeleitet wird, wird als Basisklasse bezeichnet. Dies ist aber nur eine temporäre Bezeichnung, denn eine Basisklasse kann sich natürlich von einer anderen Klasse ableiten und spielt in dieser Konstellation die Rolle der abgeleiteten Klasse.

Warum soll man überhaupt eine vorhandene Klasse ableiten? Um neue Members hinzufügen zu können und die Klasse damit etwas besser handhabbar zu machen, oder um vorhandene Members der Basisklasse zu überschreiben.

▶ **Hinweis** Eine Klasse kann nur von einer anderen Klasse abgeleitet werden (Einfachvererbung). Da sich die Basisklasse immer von einer anderen Klasse, zum Beispiel *System.Object*, ableitet, kann eine Klasse indirekt mehrere Basisklassen besitzen. Sie übernimmt dabei alle Mitglieder aller ihrer Basisklassen.

Bei der PowerShell wird eine Klasse von einer anderen Klasse abgeleitet, indem bei der Definition der Klasse der Name der Basisklasse, getrennt durch einen Doppelpunkt, auf den Namen der Klasse folgt.

Beispiel

Das folgende Beispiel definiert zwei Klassen: Eine Klasse *UserData*, die die Rolle der Basisklasse spielt, und die mit *UserName* und *UserId* über zwei Members verfügt, und eine Klasse *DomainUserData*, die sich von der Klasse *UserData* ableitet und mit *DomainName* über ein eigenes Member verfügt.

```
<#
 .Synopsis
 Abgeleitete Klassen
#>

class UserData
{
  [Int]$UserId

  UserData()
  {
    $this.UserId = -1
  }

  UserData([Int]$UserId)
  {
    $this.UserId = $UserId
  }

}

class DomainUserData : UserData
{
  [String]$DomainUserName
}
```

Wird aus der Klasse *DomainUserData* ein Objekt gemacht, stehen über die Variable die Eigenschaften der Klassen *UserData* und *DomainUserData* zur Verfügung.

Beispiel

Das folgende Beispiel macht aus der Klasse *DomainUserData* ein Objekt und weist den Eigenschaften *UserName*, *UserId* und *DomainName* jeweils Werte zu.

```
$user1 = [DomainUserData]::new()
$user1.UserId = 1000
$user1.Username = "Hubert K."
$user1.DomainName = "pskurs.local"
```

Auch wenn die Eigenschaften *UserId* und *UserName* in der Klasse *UserData* definiert wurden, spielt dies für das Verwenden der Eigenschaften keine Rolle.

Auch über das Abfragen der Eigenschaft lässt sich nicht feststellen, ob eine Eigenschaft in der aktuellen Klasse oder einer Basisklasse definiert wurde.

Interessant ist, dass die Vererbungshierarchie über die *psobject*-Eigenschaft, über die bei PowerShell jedes Objekt verfügt, und deren *TypeNames*-Eigenschaft sichtbar wird.

Beispiel

Der folgende Befehl gibt die Namen der Klassen (Typen) aus, von denen sich die Klasse *DomainUserData* ableitet.

```
$user1 = [DomainUserData]::new()
$user1.psobject.TypeNames
DomainUserData
UserData
System.Object
```

Die Klasse *DomainUserData* leitet sich von der Klasse *UserData* ab, die sich wiederum von der Klasse *System.Object* ableitet. Wem solche Ausgaben in der Vergangenheit eventuell etwas kryptisch erschienen sein mögen, sobald man das im Grunde relativ einfache Prinzip der Vererbung verstanden hat, ergibt die Ausgabe der „Vererbungshierarchie“ auf einmal einen tieferen Sinn.

3.10.1 Ableiten mit einem Konstruktor, der nicht parameterlos ist

An Grenzen stößt das Ableitungsprinzip bei Klassen aktuell immer dann, wenn die Basisklasse keinen Konstruktor besitzt, der ohne Parameter aufgerufen werden kann.

Beispiel

Das folgende Beispiel zeigt eine Ableitung, die in der aktuellen Version der PowerShell nicht funktioniert. Der Grund ist, dass die Basisklasse *UserData* nur einen Konstruktor besitzt, der einen Parameterwert erwartet. Da mit dem Anlegen der abgeleiteten Klasse automatisch auch eine Instanz der Basisklasse angelegt wird und es bei der aktuellen Version offenbar keine Möglichkeit gibt, den Konstruktorparameter der Basisklasse zu übergeben, kommt es zu einer Fehlermeldung.

```
<#
 .Synopsis
  Ableiten von einer Klasse, die keinen parameterlosen Konstruktor
besitzt - geht bei 5.1 nicht
#>

class UserData
{
  [Int]$UserId
  UserData([Int]$UserId)
  {
    $this.UserId = $UserId
 }
}

class DomainUserData : UserData
{
  [String]$DomainUserName
}
```

Da grundsätzlich nichts dagegen sprechen sollte, in die Basisklasse einen zweiten, parameterlosen Konstruktor einzufügen, lässt sich das kleine „Problem“ auf diese Weise am einfachsten aus der Welt schaffen.

3.10.2 Ableiten von Klassen der .NET-Klassenbibliothek und der PowerShell-Bibliotheken

Als Basisklasse für eine PowerShell-Klasse kommt auch jede Klasse der .NET-Klassenbibliothek und der PowerShell-Assemblys wie zum Beispiel *System.Management.Automation.dll* in Frage. Es gibt zwei Einschränkungen: Die Klasse muss „public" sein und sie darf nicht intern mit dem Modifizierer *sealed* definiert worden sein. Ein Beispiel für eine solche „versiegelte" Klasse ist *PSCredential*, die sich daher leider nicht ableiten lässt.

Ob eine Klasse „sealed" ist, lässt sich am einfachsten herausfinden, indem man versucht, von dieser Klasse abzuleiten. Ist es nicht möglich, ist eine entsprechende Fehlermeldung die Folge.

Beispiel

Der folgende Befehl gibt die Namen aller versiegelten Klassen der PowerShell-Assembly *System.Management.Automation* nach den Namen der Klassen sortiert aus.

```
[PsObject].Assembly.GetTypes()| Where-Object IsSealed | Select-Object
Name | Sort-Object Name
```

Beispiel

Der folgende Befehl gibt die Namen aller Klassen der PowerShell-Assembly *System.Management.Automation* sortiert aus, die öffentlich und nicht versiegelt sind und von denen sich daher eine Klasse ableiten lässt.

```
[PsObject].Assembly.GetTypes()| Where-Object { $_.IsPublic –and
!$_.IsSealed } | Select-Object Name | Sort-Object Name
```

Die Namen der Klassen sind so speziell, dass bereits eine entsprechend spezielle Anforderung vorliegen müsste, um tatsächlich eine interne PowerShell-Klasse in einem Skript abzuleiten.

3.10.3 Members überschreiben

Ein weiterer kleiner Vorteil des Ableitens ist, dass in der abgeleiteten Klasse einzelne Member der Basisklasse durch eine neue Implementierung ersetzt werden können. Diese Technik wird Überschreiben (engl. „overiding") genannt und darf nicht mit dem Überladen verwechselt werden. Ein Member zu überschreiben bedeutet, in einer abgeleiteten Klasse eine Eigenschaft oder eine Methode der Basisklasse durch eine neue Implementierung zu ersetzen. Ein Beispiel wäre eine Remove-Methode, die in der Basisklasse eine andere Wirkung hat als in der abgeleiteten Klasse. Das Überschreiben unterscheidet sich vom Überladen eines Members dadurch, dass es nur in einer abgeleiteten Klasse möglich ist. Die überschriebene Methode muss denselben Namen und dieselben Parameter verwenden wie die Methode der Basisklasse, die überschrieben wird. Eine Eigenschaft muss denselben Namen und denselben Datentyp verwenden wie die Eigenschaft in der Basisklasse.

Beispiel

In dem folgenden Beispiel wird in der Klasse *UserData* die Methode „ToString()“ überschrieben, die die Klasse von ihrer Basisklasse *System.Object* erbt. Die neue „ToString()“-Methode gibt Informationen über ein Objekt aus, die sich aus den Werten von einzelnen Eigenschaften zusammensetzt.

```
<#
 .Synopsis
 Überschreiben einer Methode einer Basisklasse
#>

class UserData
{
  [Int]$UserId

  UserData([Int]$UserId)
  {
    $this.UserId = $UserId
  }

  [string]ToString()
  {
    return "UserID=$($this.UserId)"
  }
}
```

Zuerst wird ein Objekt angelegt:

```
$User1 = [UserData]::new(123)
```

Es ist interessant, wie sich die Ausgabe des Objekts außerhalb und innerhalb einer Zeichenkette unterscheidet. Wird die Variable *$User1* als Teil einer Zeichenkette ausgegeben, wird automatisch die *ToString()*-Methode ausgeführt. Wurde diese nicht überschrieben, wird lediglich die Typenbezeichnung des Objekts ausgegeben, ansonsten jener Wert, der von der überschriebenen Methode zurückgegeben wird.

Gibt es denn noch weitere Methoden bei der *System.Object*-Klasse, die in einer abgeleiteten Klasse überschrieben werden könnten? Im Prinzip ja. Die Namen der Methoden lassen sich einfach herausfinden.

Beispiel

Der folgende Befehl gibt alle Methoden-Members der Object-Klasse aus.

```
[Object].GetMethods().Name
```

Es sind die Methoden *GetType*, *Equals*, *ToString*, *ReferenceEquals* und *GetHashCode*.

Eine weitere Methode, die für ein Überschreiben in Frage käme, ist die *GetType()*-Methode, die jenes *RuntimeType*-Objekt zurückgibt, das den Typ repräsentiert, auf dem das Objekt basiert. Hier wäre es denkbar, stattdessen eine Zeichenkette mit der Typenbezeichnung zurückzugeben.

Beispiel

Das folgende Beispiel definiert eine Klasse *UserData*, in der die „GetType()"-Methode überschrieben wurde. Doch Vorsicht: Das Beispiel wird nicht funktionieren und die ISE wird vermutlich abstürzen (eine Neuinstallation von Windows wird allerdings nicht erforderlich sein).

```
<#
 .Synopsis
 Überschreiben einer Get-Methode einer Basisklasse mit "Absturz"
#>

class UserData
{
  [Int]$UserId

  UserData()
  {
  }

  [Object]GetType()
  {
    return "Typename = $($this.GetType().FullName)"
  }
}

class DomainUserData : UserData
{
  [String]$DomainUserName

  DomainUserData([Int]$UserId)
  {
    $this.UserId = $UserId
  }

}

$user1 = [DomainUserData]::new(100)
$User1.DomainUserName = "Hannes P."
$user1.GetType()
```

Verantwortlich für den „Crash" sind die beiden Befehle:

```
$user1 = [UserData]::new()
$user1.GetType()
```

Eigentlich soll doch nur die Typenbezeichnung des Objekts durch den Aufruf der *GetType()*-Methode und Ansprechen der *FullName*-Eigenschaft des resultierenden RuntimeType-Objekts ausgegeben werden. Was ist bei dem letzten Beispiel falsch?

Der Fehler liegt in dem folgenden, vollkommen harmlosen Ausdruck:

```
$TypName = $this.GetType().FullName
```

Haben Sie ihn entdeckt? Wenn ja, dann herzlichen Glückwunsch, da der Fehler zu jener Sorte gehört, die so offensichtlich ist, dass sie dadurch schwer zu entdecken sind. In dem letzten Befehl ruft sich die *GetType()*-Methode selber auf, was zu einer endlosen Aufrufkette und damit irgendwann zum Absturz der ISE führt.

Was im obigen Beispiel aufgerufen werden soll, ist die *GetType()*-Methode des Objekts der Basisklasse, das mit dem Anlegen des *UserData*-Objekts ebenfalls angelegt wird, konkret jenes Objekts vom Typ *Object*. Doch wie sagt man der PowerShell „Ruf nicht die Methode der aktuellen Klasse, sondern die Methode der Basisklasse auf“? Ganz einfach, indem die Instanz der Klasse, repräsentiert durch die Variable *$this*, auf die bei der PowerShell üblichen Art und Weise in den Typ der Basisklasse konvertiert wird. Konkret: *$this* muss daher lediglich ein *[Object]* vorangestellt werden.

```
[String]GetType()
{
  $TypName = (([Object]$this)).GetType().FullName
  return "Die Typenbezeichnung lautet: $TypName"
}
```

Beispiel

Das folgende Beispiel überschreibt erneut die *GetType()*-Methode der *Object*-Klasse, dieses Mal aber richtig, indem durch eine Typenkonvertierung die *GetType()*-Methode der *Object*-Klasse aufgerufen wird.

```
<#
 .Synopsis
 Überschreiben einer GetType-Methode der Basisklasse ohne "Absturz"
#>

class UserData
{
  [Int]$UserId

  UserData()
  {
  }

  [String]GetType()
  {
    return "Typename = $(([Object]$this).GetType().FullName)"
  }

}

class DomainUserData : UserData
{
  [String]$DomainUserName

  DomainUserData([Int]$UserId)
  {
    $this.UserId = $UserId
  }
}
```

Beispiel

Der Aufruf der überschriebenen *GetType()*-Methode sieht wie folgt aus.

```
$user1 = [UserData]::New()
$user1.GetType()
```

3.11 Typenformatierung mit eigenen Klassen

Die über das *Update-FormatData*-Cmdlet durchgeführte Definition eigener Formate für die Ausgabe bestimmter Objekttypen per *Format-Table* und *Format-List* lässt sich natürlich auch mit jenen Typen durchführen, die per *class*-Befehl definiert wurden. Dazu wird lediglich eine Formatdefinitionsdatei benötigt. Das ist eine Textdatei, welche die Erweiterung „.format.ps1xml“ tragen muss, und deren Pfad beziehungsweise Name beim *Update-FormatData*-Cmdlet auf den *PrependPath*-Parameter folgt.

Beispiel

Das folgende Beispiel ist etwas umfangreicher, denn es enthält neben der Klassendefinition der Klasse *UserData* auch den XML-Text für die Formatierungsinformation für den Typ „UserData“. Die Formatierungsdefinition definiert eine View mit dem Namen „Standard“, die den Aufbau der Tabelle (*TableControl*-Element) festlegt, in der für jede Spalte unter anderem ihre Breite, die Ausrichtung und der Name der Überschrift festgelegt wird. Außerdem kann der Inhalt einer Spalte individuell festgelegt werden.

```
<#
 .Synopsis
 Individuelle Typenformatierung für Objekte des Typs UserData
#>

# Definition einer Tabellenformatierung für den Typ UserData
$UserDataFormatDef = @'
<Configuration>
  <ViewDefinitions>
    <View>
      <Name>Standard</Name>
      <ViewSelectedBy>
         <TypeName>UserData</TypeName>
       </ViewSelectedBy>
       <TableControl>
          <TableHeaders>
            <TableColumnHeader>
              <Label>User-Id</Label>
              <Width>10</Width>
            </TableColumnHeader>
            <TableColumnHeader>
              <Label>Name</Label>
              <Width>16</Width>
            </TableColumnHeader>
            <TableColumnHeader>
              <Label>User-Typ</Label>
              <Width>12</Width>
            </TableColumnHeader>
            <TableColumnHeader>
              <Label>Domäne</Label>
              <Width>24</Width>
            </TableColumnHeader>
          </TableHeaders>
          <TableRowEntries>
            <TableRowEntry>
              <TableColumnItems>
                <TableColumnItem>
                    <ScriptBlock>$_.Id</ScriptBlock>
                </TableColumnItem>
                <TableColumnItem>
                    <ScriptBlock>$_.UserName</ScriptBlock>
                </TableColumnItem>
```

```
                <TableColumnItem>
                    <ScriptBlock>switch ($_.UserType) { "Domain" {
"Domäne"} default {"Keine" }}</ScriptBlock>
                </TableColumnItem>
                <TableColumnItem>
                    <ScriptBlock>$_.DomainName</ScriptBlock>
                </TableColumnItem>
              </TableColumnItems>
            </TableRowEntry>
          </TableRowEntries>
      </TableControl>
    </View>
  </ViewDefinitions>
</Configuration>
'@

# Speichern des Xml in eine Textdatei
$FormatPath = Join-Path -Path $PSScriptRoot -ChildPath
"UserData.format.ps1xml"
$UserDataFormatDef | Set-Content -Path $FormatPath

# Aktualisieren der Typenformatierung der PowerShell
Update-FormatData -PrependPath $FormatPath

# Definition eines Enum
enum UserType
{
  Domain
  Local
}

# Definition der Klasse UserData
class UserData
{
  [String]$UserName
  [UserType]$UserType
  [Int]$Id

  UserData()
  {
    $this.Id = -1
    $this.UserType = "Local"
  }
}

# Definition der Klasse DomainUserData
class DomainUserData : UserData
{
  [String]$DomainName

  DomainUserData([Int]$Id)
  {
    $this.Id = $Id
    $this.UserType = "Domain"
  }
}
```

Beispiel

Die folgende Befehlsfolge legt ein Objekt vom Typ *DomainUserData* an.

```
$user1 = [DomainUserData]::new(100)
$user1.UserName = "Erwin L."
$user1.DomainName = "pskurs.local"
$user1
```

Beispiel

Die folgende Befehlsfolge legt ein Objekt vom Typ *UserData* an.

```
# Anlegen eines Objekts vom Typ UserData
$user2 = [UserData]::new()
$user2.UserName = "Paul P."
$user2
```

3.12 Zusammenfassung

Der mit der PowerShell Version 5.0 eingeführte *class*-Befehl erlaubt das Definieren eigener Klassen und damit von Typen, die innerhalb eines Skripts für das Definieren von Objekten genutzt werden können. Eine Klasse kann mit Members ausgestattet werden, Konstruktoren besitzen und sich von einer anderen Klasse ableiten. Members können innerhalb einer Klasse überladen und in der abgeleiteten Klasse überschrieben werden. Aus einer Klasse werden entweder per *New-Object*-Cmdlet oder mit der statischen *new*-Methode der Klasse Objekte gemacht.

Es gibt im Wesentlichen drei Gründe dafür, Klassen zu verwenden:

1. DSC-Ressourcen werden mit einer Klasse mit deutlich weniger Aufwand erstellt als ohne die Verwendung von Klassen. Nachteil: Die Ressource kann nicht unter WMF 4.0 in eine MOF-Datei kompiliert werden.
2. Ein Skript wird durch Klassen übersichtlicher und etwas sauberer strukturiert, indem Befehlsfolgen und Variablen, die eine bestimmte Aufgabe übernehmen, in einer Klasse zusammengefasst werden. Anders als bei Functions müssen Methoden innerhalb einer Klassendefinition ihren Wert mit dem *return*-Befehl zurückgeben.
3. Mit der Möglichkeit, Klassen definieren zu können, zieht die PowerShell mit praktisch allen anderen Skript- und Programmiersprachen gleich und wird damit für Entwickler etwas attraktiver. Einen class-Befehl gab es bereits bei VBScript, das lange vor der PowerShell im Rahmen des Windows Scripting Hosts für das Erstellen von Skripten verwendet wurde.

Functions für Fortgeschrittene

4

Zusammenfassung

In diesem Kapitel wird beschrieben, welche Möglichkeiten mit „Advanced Functions" einhergehen. Dabei geht es im Kern um die erweiterten Parameterdeklarationen, das Prinzip der Parameterbindung, um eine flexible Parametervalidierung und um Functions, die genau wie Cmdlets die Pipeline abarbeiten.

4.1 Was macht eine Function „Advanced"?

Der Begriff „Advanced" bezieht sich bei einer Function auf die Parameter der Function, nicht auf ihren Inhalt. Eine „Advanced Function" verwendet dieselben Mechanismen bezüglich der Parameterwert-Zuordnung, der Pipeline-Bindung und der Bestätigungsanforderung wie ein Cmdlet. Der Inhalt der Function spielt aber keine Rolle. Bereits durch die Verwendung des *CmdletBinding*-Attributs für die gesamte Parameterdeklaration und/oder des *Parameter*-Attributs für einen einzelnen Parameter wird aus einer Function eine Advanced Function.

▶ **Tipp** In der Hilfe wird das Thema unter den Themen „about_functions_advanced", „about_functions_advanced_parameters" und „about_Functions_Advanced_Methods" ausführlich dokumentiert.

4.1.1 Die Rolle der Attribute

In diesem Kapitel ist oft von „Attributen" die Rede. Ein Attribut ist allgemein eine ergänzende Information. Attribute ergänzen immer einzelne Befehle eines Programms. Bei der PowerShell können Functions als auch einzelne Parameter mit Attributen erweitert werden. Da Attribute auf Klassen basieren und damit Typen sind, werden ihre Namen in eckige

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_4

Klammern gesetzt. Ein Beispiel für ein Attribut ist das *Parameter*-Attribut. Sein Name wird in eckige Klammern gesetzt und geht im Rahmen des *param*-Befehls einem Parameter voraus.

Beispiel

Der folgende Befehl definiert den *Path*-Parameter als Pflichtparameter.

```
param([Parameter(Mandatory=$true)][String]$Path)
```

Hinter *Parameter* steht die Klasse *ParameterAttribute* im Namespace *System.Management.Automation*. Das gibt unter anderem ein „[Parameter].FullName“ aus. *Parameter* ist damit ein weiterer Typenalias.

▶ **Hinweis** Es ist eine bei der .NET-Laufzeit übliche Konvention, an den Namen des Attributs noch ein „Attribute“ anzuhängen. Die Klasse heißt daher z. B. „ParameterAttribute“ und nicht nur „Parameter“.

Beispiel

Das folgende Beispiel ist etwas spezieller. Es gibt die Namen aller Klassen in der PowerShell-Assembly aus, die mit dem Wort „Attribute“ enden und damit Attributklassen sind.

```
[PSCustomObject].Assembly.GetTypes().Where{$_.Name -like "*Attribute" } |
Sort-Object Name | Select-Object Name
```

Die Ausgabe ist relativ umfangreich und dient in erster Linie dazu zu demonstrieren, dass es viele dieser Attributklassen gibt. Nur die wenigsten haben etwas mit Parametern zu tun. Um gezielt etwas über die Parameter-Attribute zu erfahren, ist die PowerShell-Hilfe der bessere Ort.

4.1.2 Das Prinzip der Parameter-Zuordnung

Die PowerShell ordnet allen auf den Namen einer Function oder eines Skripts folgenden Werte den Parametern der Function oder des Scripts in der Reihenfolge zu, in der die Parameter festgelegt wurden. Überschüssige Werte werden ignoriert. Die einzelnen Werte, die auch als Argumente bezeichnet werden, werden nicht per Komma, sondern per Leerzeichen getrennt.

Beispiel

Ein Function *f1* besitzt drei Parameter: *P1*, *P2* und *P3*.

```
function f1
{
  param($P1, $P2, $P3)
  "P1=$P1 P2=$P2 P3=$P3"
}
```

Im einfachsten Fall folgen die Werte der Parameter auf den Namen der Function.

```
f1 10 20 30
```

Die drei Zahlen werden einzeln den drei Parametern zugeordnet, da alle drei Parameter Positionsparameter sind. Überschüssige Argumente werden ignoriert.

Natürlich können die Parameter auch benannt verwendet werden. In diesem Fall spielt die Reihenfolge der Argumente keine Rolle, da ein Argument einem Parameter über seinen Namen und nicht seine Position zugeordnet wird.

Beispiel

Das folgende Beispiel ruft die Function *f1* auf und ordnet jedem Parameter einen Wert über seinen Namen zu.

```
f1 -P3 30 -P1 10 -P2 20
```

Die Verwendung von Positionsparametern ist das Standardverhalten bei Functions und Skripten. In den folgenden Abschnitten wird gezeigt, wie sich dieses Standardverhalten ändern lässt.

4.2 Das CmdletBinding-Attribut

Das Attribut *CmdletBinding* bezieht sich auf die gesamte Parameterdeklaration und geht daher dem *param*-Befehl voraus. *[CmdletBinding]* ist ein Typenalias für die Klasse *System.Management.Automation.CmdletBindingAttribute*. Gut, dass wir ihn nicht vollständig eingeben müssen. Ein Attribut steht im Allgemeinen für mehrere Einstellungen. Diese werden als Eigenschaften der Attributklasse in runde Klammern gesetzt und erhalten durch eine Zuweisung einen Wert.

Die Eigenschaften des *CmdletBinding*-Attributs werden in Tab. 4.1 zusammengestellt.

Da Attribute auf Klassen basieren und die Einstellungen, die über ein Attribut zur Verfügung gestellt werden, statische Eigenschaften der jeweiligen Klassen sind, lassen sich die Einstellungen sehr einfach ausgeben.

Beispiel

Der folgende Befehl gibt die Namen aller Eigenschaften des *CmdletBinding*-Attributs aus.

```
[CmdletBinding].GetProperties().Name
```

▶ **Hinweis** Die Rolle des *CmdletBinding*-Attributs wird in der Hilfe unter „about_Functions_CmdletBindingAttribute“ ausführlich beschrieben.

Tab. 4.1 Die Einstellungen des CmdletBinding-Attributs

Eigenschaft	Bedeutung
ConfirmImpact	Legt den „Level“ für Bestätigungsanforderungen fest.
DefaultParametersetName	Der Name des Default-Parametersets. Es kann auch ein Leerstring eingetragen werden. Dies ist erforderlich, um zu erreichen, dass Parameter, die keinem Parameterset zugeordnet werden, in allen Parametersets enthalten sind.
HelpURI	Legt eine URI für das Abrufen einer Hilfe für das Skript oder die Function fest.
PositionalBinding	Per Voreinstellung sind alle Parameter Positionsparameter. Durch Setzen dieser Eigenschaft auf *$false* ist das nicht mehr der Fall. Über das *Parameter*-Attribut und seine *Position*-Eigenschaft kann ein einzelner Parameter wieder zum Positionsparameter gemacht werden.
SupportsPaging	Legt fest, ob die Parameter *First*, *IncludeTotalCount* und *Skip* zur Verfügung stehen und damit zum Beispiel eine Begrenzung der Ausgabe möglich ist. Diese Funktionalitäten müssen aber, zum Beispiel innerhalb einer Function, implementiert werden.
SupportsShouldProcess	Legt fest, ob die Parameter *Confirm* und *Whatif* zur Verfügung stehen.
SupportsTransactions	Legt fest, ob Transaktionen unterstützt werden.

Tab. 4.2 Die Eigenschaften des Parameter-Attributs

Eigenschaft	Bedeutung
ByPropertyName	Der Parameter kann seinen Wert über den Wert der Eigenschaft des Objekts in der Pipeline erhalten, die denselben Namen besitzt wie der Parameter oder eines Alias des Parameters.
ByValue	Der Parameter kann seinen Wert über den Wert in der Pipeline erhalten.
DontShow	Legt fest, dass der Parameter nicht als Teil der Syntax angezeigt wird.
HelpMessage	Ein Hilfetext für den Parameter.
ParametersetName	Der Name des Parametersets, zu dem der Parameter gehört.
Position	Die Position bei einem Positionsparameter. Per Voreinstellung ist jeder Parameter ein Positionsparameter mit einem Positionswert, der automatisch durchnummeriert wird.

4.3 Das Parameter-Attribut

Das *Parameter*-Attribut erweitert die Definition eines einzelnen Parameters. Es legt folgende Details fest. Tab. 4.2 stellt die Eigenschaften der *ParameterAttribute*-Klasse zusammen.

Auch beim *Parameter*-Attribut lassen sich seine statischen Eigenschaften mit einem Befehl ausgeben.

Beispiel

Der folgende Befehl gibt die Namen aller Eigenschaften des Parameter-Attributs aus.

```
[Parameter].GetProperties().Name
```

4.4 Die Parameterbindung im Detail

Parameterbindung bedeutet, dass ein Parameter seinen Wert nicht direkt, sondern indirekt aus der Pipeline bezieht. Indirekt bedeutet, dass der Wert zuvor von einem anderen Befehl dort abgelegt wurde. Die Parameterbindung ist dafür zuständig, dass der folgende Befehl funktioniert:

```
"AudioSrv" | Get-Service
```

Warum dieser Befehl funktioniert, soll im Folgenden erklärt werden. Es gibt zwei Sorten der Parameterbindung:

1. Über den Inhalt der Pipeline.
2. Über den Wert einer Eigenschaft des Objekts, das sich in der Pipeline befindet.

Welche Form der Parameterbindung verwendet wird, hängt von dem Parameter ab. Eine Situation, in der die Parameterbindung eine Rolle spielt, ist, wenn ein Command auf den Pipe-Operator folgt und dem Pflichtparameter des Commands kein Wert direkt übergeben wurde. In diesem Fall versucht die PowerShell, dem Parameter den Wert aus der Pipeline zuzuordnen. Gelingt dies nicht, ist eine Fehlermeldung die Folge. Eine andere Situation liegt vor, wenn mehrere Parameter eine Pipeline-Bindung über den Wert einer Eigenschaft unterstützen. Erhalten diese Parameter keine direkten Werte, versucht die PowerShell das Objekt, das sich aktuell in der Pipeline befindet, heranzuziehen. In diesem Fall kommt es auf die Eigenschaften des Objekts und ihre Namen an, da die Zuordnung auf der Übereinstimmung zwischen Parameternamen und den Namen einer Eigenschaft basiert.

Ob und welche Art der Parameterbindung ein Parameter unterstützt, erfahren Sie aus der Hilfe zu dem Parameter.

Beispiel

Der folgende Befehl ruft die Beschreibung zum Parameter *Name* des Set-Service-Cmdlets ab.

```
Get-Help Get-Service -Parameter Name

-Name [<String[]>]
    Specifies the service names of services to be retrieved. Wildcards
are permitted. By default, this cmdlet gets all of the services on the
computer.

    Erforderlich?               False
    Position?                   1
    Standardwert                none
    Pipelineeingaben akzeptieren?true(ByValue, ByPropertyName)
    Platzhalterzeichen akzeptieren?false
```

Wichtig ist die Angabe „Pipelineeing.aben akzeptieren?“, denn sie gibt an, ob und auf welche Weise der Parameter seinen Wert aus der Pipeline bezieht. Der *Name*-Parameter,

um den es in diesem Beispiel geht, unterstützt beide Formen der Parameterbindung: Über den Inhalt der Pipeline (*ByValue*) und über den Namen einer Eigenschaft des Objekts in der Pipeline (*ByPropertyName*).

Die Parameterbindung über den Inhalt der Pipeline bei Get-Service wurde bereits demonstriert. Diese Variante ist in der Praxis eher die Ausnahme. Sehr viel häufiger ist die Parameterbindung über den Namen der Eigenschaft.

Beispiel

Der folgende Befehl beendet einen Systemdienst per *Stop-Service*-Cmdlet, der zuvor über ein *Get-Service*-Cmdlet geholt wurde.

```
Get-Service –Name AudioSrv | Stop-Service
```

Es fällt auf, dass auf *Stop-Service* kein Parameter folgt. Der Grund dafür ist, dass der Name-Parameter als Pflichtparameter seinen Wert aus der Pipeline bezieht. Konkret: Der Parameter erhält den Wert der *Name*-Eigenschaft des *ServiceController*-Objekts, das sich in der Pipeline befindet.

4.4.1 Wenn die Parameterbindung nicht funktioniert

Die Parameterbindung ist ein Mechanismus, der nicht immer funktionieren muss. Ein Beispiel ist das Beenden eines Prozesses per *Stop-Process*-Cmdlet, wenn dem Cmdlet lediglich der Name des Prozesses übergeben wird.

Beispiel

Das Beispiel geht davon aus, dass ein Prozess mit dem Namen „Mspaint“ ausführt. Der folgende Befehl schlägt fehl:

```
"mspaint" | Stop-Process
```

Der Grund für die resultierende Fehlermeldung ist einfach. Anders als zum Beispiel bei *Get-Service* unterstützt der *Name*-Parameter entweder keine Parameterbindung oder keine „ByValue“-Bindung. Ein Blick in die Hilfe verrät, dass Letzteres der Fall ist. Es wird nur die Parameterbindung „ByPropertyName“ unterstützt. In der Praxis ist das kein Problem, da das Cmdlet eher in der Form „Stop-Process – Name MsPaint“ aufgerufen wird, die Pipeline und damit auch die Parameterbindung in dieser Schreibweise keine Rolle spielt.

Möchte man unbedingt die Pipeline verwenden, gibt es auch dafür eine Lösung. Gesucht ist ein beliebiges Objekt, das eine *Name*-Eigenschaft besitzt, deren Wert der Name des zu beendenden Prozesses ist. Zu Anschauungszwecken kann man ein solches Objekt auch selber herstellen. Die Details werden in Kap. 10 „Aus Text werden Objekte“ verraten.

Beispiel

Der folgende Befehl legt ein Objekt mit einer *Name*-Eigenschaft und dem Wert „mspaint“ an und übergibt dieses dem *Stop-Process*-Cmdlet per Pipe-Operator.

```
[PsCustomObject]@{Name="Mspaint" } | Stop-Process
```

Dieses Mal wird der Prozess beendet. Auch wenn der Befehl etwas ungewöhnlich ist, er funktioniert tadellos, da es dem *Stop-Process*-Cmdlet egal ist, welche Sorte von Objekt sich in der Pipeline befindet. Wichtig ist nur, dass dieses Objekt eine *Name*-Eigenschaft besitzt.

4.4.2 Die Parameterbindung sichtbar machen

Wer hinter die Kulissen schauen und nachvollziehen möchte, welche Schritte die PowerShell intern unternimmt, um einem Parameter einen Wert zuzuordnen, kann dies über das *Trace-Command*-Cmdlet tun. Eine Warnung gleich vorweg: Der Output des Cmdlets ist sehr umfangreich, da wirklich jedes Detail dokumentiert wird.

Beispiel

Der folgende Befehl untersucht eine nicht funktionierende Parameterbindung mit dem *Trace-Command*-Cmdlet. Das Protokoll wird sowohl ausgegeben als auch in eine Textdatei geschrieben.

```
Start-Process -FilePath mspaint
$SB = { "mspaint" | Stop-Process }
Trace-Command -Name ParameterBinding -PSHOst -Expression $SB -FilePath
C:\PSTraceLog.txt
```

4.5 Functions mit Pipeline-Parametern

Derselbe Mechanismus, der bei den Parametern eines Cmdlets dafür sorgt, dass diese ihren Wert aus der Pipeline beziehen können, lässt sich natürlich auch auf die Parameter einer Function oder eines Skripts anwenden. Die Function oder das Skript kann damit genau wie ein Cmdlet pipelinebasierend ausgeführt werden. Dafür ist das *Parameter*-Attribut mit seinen Eigenschaften *ValueFromPipeline* und *ValueFromPipelineByPropertyName* zuständig. Die Techniken, die im Rahmen des nächsten Beispiels vorgestellt werden, lassen sich genauso auf eine Function wie auf ein Skript anwenden.

Beispiel

Das folgende Beispiel ist etwas umfangreicher, da eine Function vorgestellt wird, deren Parameter *Path* seinen Wert auch aus der Pipeline beziehen soll. Die Function wird dazu Schritt für Schritt erweitert, bis das gewünschte Ergebnis resultiert. Das Beispiel

umfasst eine Function, die die Speicherkosten für ein Verzeichnis berechnet. Sie besitzt mit *Path* und *KostenMB* zwei Parameter. Eine Pipeline-Bindung ist noch nicht möglich.

```
function Get-Speicherkosten
{
  param([Parameter(Mandatory=$true)][String]$Path,
[Double]$SpeicherkostenMB)

  $SummeBytes = 0
  Get-ChildItem -Path $Path -Directory -Recurse | ForEach-Object {
    Get-ChildItem -Path $_.FullName | ForEach-Object {
      $SummeBytes += $_.Length
      }
  }
  $SpeicherkostenGesamt = [Math]::Round($SummeBytes / 1MB *
$SpeicherkostenMB, 3)
  [PSCustomObject]@{Pfad=$Path;Kosten=$SpeicherkostenGesamt}
}
```

In der aktuellen Version kann keiner der beiden Parameter seinen Wert aus der Pipeline erhalten. Die Werte müssen daher wie üblich auf den Namen der Function folgen. Das soll im Folgenden für den *Path*-Parameter geändert werden.

Beispiel

Das folgende Beispiel zeigt eine Function, deren *Path*-Parameter seinen Wert aus der Pipeline beziehen kann.

```
function Get-Speicherkosten
{
  param([Parameter(Mandatory=$true,
   ValueFromPipelineByPropertyName=$true)][String]$Path,
   [Double]$SpeicherkostenMB)

  $SummeBytes = 0
  Get-ChildItem -Path $Path -Directory -Recurse | ForEach-Object {
    Get-ChildItem -Path $_.FullName | ForEach-Object {
        $SummeBytes += $_.Length
        }
  }

  $SpeicherkostenGesamt = [Math]::Round($SummeBytes / 1MB *
$SpeicherkostenMB, 3)
  [PSCustomObject]@{Pfad=$Path;Kosten=$SpeicherkostenGesamt}
}
```

Der folgende Aufruf sollte jetzt funktionieren:

```
dir -Path $PSHOME -Directory | Get-Speicherkosten -SpeicherkostenMB 1
```

Doch weit gefehlt. Anstatt die Speicherkosten auszugeben, sind „jede Menge“ Fehlermeldungen (pro Verzeichnis eine) die Folge. Die Parameterbindung hat also noch nicht funktioniert. Der Grund dafür ist einfach: Das Objekt, das vom *Get-ChildItem*-Cmdlet in die Pipeline gelegt wird, besitzt keine Name-Eigenschaft. Jene Eigenschaft des Objekts in der Pipeline, die für den Verzeichnispfad steht, heißt „FullName“. Damit kann die gewünschte „ByProperty“-Bindung nicht durchgeführt werden.

Die einfachste Lösung wäre natürlich, den Parameter von „Path“ in „Fullname“ umzubenennen. Falls man dies aus irgendeinem nicht möchte, gibt es eine weitere einfache Lösung. Auch Parameter können Aliase besitzen. Dafür ist das *[Alias]*-Attribut zuständig. Der oder die Name(en) des Parameters folgen in Klammern. Geben wir dem Parameter *Path* daher einfach den Alias „FullName“, damit die Parameterbindung über den Namen der Eigenschaft funktioniert.

Beispiel

Das folgende Beispiel erweitert den Parameter *Path* um einen Alias „Fullname“, so dass dieser an die Eigenschaft *Fullname* gebunden werden kann.

```
function Get-Speicherkosten
{
  param([Alias("FullName")][Parameter(Mandatory=$true,
ValueFromPipelineByPropertyName=$true)][String]$Path,
        [Double]$SpeicherkostenMB)
```

Damit funktioniert der folgende Aufruf:

```
dir -Path $PSHOME -Directory | Get-Speicherkosten -SpeicherkostenMB 1
```

Einen Kritikpunkt gibt es nach wie vor. Es erfolgt auch dann nur eine Ausgabe, wenn durch den *dir*-Befehl mehrere Verzeichnisse geliefert und damit mehrere Objekte nacheinander in die Pipeline gelegt werden. Wenn man genau hinsieht, wird man feststellen, dass nur die Größe des letzten Verzeichnisses ausgegeben wird, das von *Get-ChildItem* übergeben wurde. Die Function wurde damit nur einmal ausgeführt. Was noch fehlt ist, dass unsere Function auch die Pipeline abarbeiten kann und sooft wiederholt wird wie Objekte in die Pipeline gelegt werden.

4.6 Functions, die die Pipeline abarbeiten

Soll eine Function die Pipeline abarbeiten, bedeutet das konkret, dass die Befehle der Function für jedes Objekt, das über die Pipeline übergeben wird, einmal ausgeführt werden. Damit das möglich wird, muss die Function einen *Process*-Befehl enthalten, auf den ein Scriptblock folgt. Die Befehle dieses Scriptblocks werden automatisch pro Objekt in der Pipeline einmal ausgeführt. Wie nicht anders zu erwarten war, wird der Inhalt der Pipeline in dem Scriptblock über die Spezialvariable *$_* angesprochen.

Wenn eine Function einen *Process*-Block enthält, darf sie keine anderen Befehle außerhalb dieses Blocks mehr enthalten. Alle Befehle, die nur einmal ausgeführt werden sollen, werden entweder in den *Begin*- oder in den *End*-Block eingegeben. Beide Blöcke werden nur einmal ausgeführt: der *Begin*-Block zu Beginn der Pipeline-Verarbeitung, der *End*-Block entsprechend am Ende der Pipeline-Verarbeitung.

Beispiel

Die folgende Function erweitert die Function *Get-Speicherkosten* um einen Process-Block, damit die Pipeline vollständig abgearbeitet wird.

```
function Get-Speicherkosten
{
  param([Alias("FullName")]
  [Parameter(Mandatory=$true, ValueFromPipelineByPropertyName=$true)]
  [String]$Path,
  [Double]$SpeicherkostenMB)

  begin
  {
    $SummeBytes = 0
  }

  process
  {
    Get-ChildItem -Path $Path -Directory -Recurse | ForEach-Object {
    Get-ChildItem -Path $_.FullName | ForEach-Object {
        $SummeBytes += $_.Length
        }
    }
    $SpeicherkostenGesamt = [Math]::Round($SummeBytes / 1MB *
      $SpeicherkostenMB, 3)
    [PSCustomObject]@{Pfad=$Path;Kosten=$SpeicherkostenGesamt}
  }

  end { }
}
```

Damit ist endlich auch der folgende Aufruf möglich, bei dem Größe und Speicherkosten für jedes Verzeichnis ausgegeben werden, das über die Pipeline übergeben wird.

```
dir -Path $PSHOME -Directory | Get-Speicherkosten -SpeicherkostenMB 1
```

4.7 Functions mit einer eingebauten Bestätigungsanforderung

Das Hinzufügen des *CmdletBinding*-Attributs für alle Parameter oder das Hinzufügen des *Parameter*-Attributs für einen einzelnen Parameter führt dazu, dass die allgemeinen Parameter wie *Verbose* oder *ErrorAction* zur Verfügung stehen. Die Parameter *Confirm* und *WhatIf* sind nicht dabei. Soll eine Function oder ein Skript die Parameter *Confirm* und *WhatIf* zur Verfügung stellen, geschieht dies nicht von alleine. Benötigt wird das *CmdletBinding*-Attribut, bei dem zwei Eigenschaften eine Rolle spielen: *SupportsShouldProcess* und *ConfirmImpact*. Während *SupportsShouldProcess* mit einem *$true*-Wert festgelegt, dass die Parameter *Confirm* und *WhatIf* angeboten werden, bestimmt *ConfirmImpact* in Abhängigkeit des Wertes der globalen Variablen *ConfirmPreference*, ob auch dann eine Bestätigungsanforderung ausgegeben wird, wenn der *Confirm*-Parameter nicht gesetzt wurde.

Der Hintergrund dieser Thematik ist etwas vielschichtiger, aber nicht kompliziert. Wir gehen im Folgenden alle Aspekte, die in diesem Zusammenhang eine Rolle spielen, der Reihe nach mit kleinen Beispielen durch. Wie immer gelten alle Hinweise sowohl für Functions als auch für Skripte.

4.7.1 Die Auswirkung des Confirm-Parameters

Wird bei einer Function der *Confirm*-Parameter gesetzt, wird dieser für alle Commands, die einen *Confirm*-Parameter besitzen, aktiviert und es wird für jedes Command eine eigene Bestätigungsanforderung eingeholt.

4.7.2 Die Auswirkung des WhatIf-Parameters

Wird bei einer Function der *WhatIf*-Parameter gesetzt, wird dieser für alle Commands, die einen *WhatIf*-Parameter besitzen, aktiviert. Jedes dieser Commands wird dadurch nicht ausgeführt, sondern es wird lediglich eine Meldung ausgegeben.

4.7.3 Implementieren von Confirm und WhatIf an einem Beispiel

In diesem Abschnitt wird eine kleine Function vorgestellt, die aus einem XML-Baum einen Knoten entfernt, wenn dieser ein bestimmtes Kriterium erfüllt. Damit die Function etwas sicherer wird, erhält sie die Parameter *Confirm* und *WhatIf*, so dass das Entfernen auf Wunsch nur nach einer expliziten Bestätigung durchgeführt wird.

Beispiel

Das XML ist für alle der folgenden Beispiele gleich aufgebaut. Es besteht aus einem Stammelement und mehreren Kindelementen.

```
$xml = @'
  <books>
    <book id='1000'>
      <title>Alles klar mit PowerShell</title>
    </book>
    <book id='1001'>
      <title>Viel Spaß mit PowerShell</title>
    </book>
    <book id='1002'>
      <title>Noch mehr Spaß mit PowerShell</title>
    </book>
  </books>
'@
```

In der ersten Version entfernt die Function einen Knoten, wenn dessen *Id*-Attribut den Wert besitzt, der über den Parameter *NodeId* festgelegt wurde.

```
function Remove-XmlNodeById
{
  param([String]$Xml, [String]$NodeId)
  Add-Type -AssemblyName System.Xml.Linq
  $Root = [System.Xml.Linq.XDocument]::Parse($Xml)
  # Es kommt hier auf die Groß-/Kleinschreibung an!
  $Node = $Root.Descendants("book") | Where-Object {
$_.Attribute("id").Value -eq $NodeId }
  $Node.Remove()
  $Root.ToString()
}
```

Da die Function kein *[CmdletBinding]*-Attribut verwendet, gibt es auch keinen *Confirm*-Parameter und der Knoten wird immer entfernt. Das soll mit der nächsten Version geändert werden.

Die nächste Version der Function *Remove-XmlNodeById* besitzt ein *[CmdletBinding]*-Attribut mit einem „SupportsShouldProcess=$true“ und damit auch die Parameter *Confirm* und *WhatIf*. Da im Rahmen der Function für das Entfernen des Knotens aber kein Command ausgeführt wird, das selber einen *Confirm*-Parameter besitzt, muss die Abfrage über die Variable *PsCmdlet* und deren *ShouldProcess*-Methode durchgeführt werden.

```
function Remove-XmlNodeById
{
  [CmdletBinding(SupportsShouldProcess=$true)]
  param([String]$Xml, [String]$NodeId)
  Add-Type -AssemblyName System.Xml.Linq
  $Root = [System.Xml.Linq.XDocument]::Parse($Xml)
  # Es kommt hier auf die Groß-/Kleinschreibung an!
  $Node = $Root.Descendants("book") | Where { $_.Attribute("id").Value -
eq $NodeId }
  # Das Entfernen soll auf Wunsch eine Bestätigungsanforderung ausgeben
  if ($PSCmdlet.ShouldProcess($Node, "Knoten mit Id $Id entfernen?"))
  {
    $Node.Remove()
  }
  $Root.ToString()
}
```

Wird die Function mit dem *Confirm*-Parameter aufgerufen, erscheint vor dem Aufruf der *Remove()*-Methode eine Bestätigungsanforderung. Für den Parameter *Target* von *ShouldProcess()* kann ein beliebiger Wert übergeben werden. Im Allgemeinen wird hier das Objekt übergeben, um das es bei der Aktion geht. Der zweite Parameter ist optional. Hier kann der Text für die Bestätigungsanforderung festgelegt werden. Wird noch ein drittes Argument, ein Leerstring genügt, übergeben, wird dieses als Überschrift für die Messagebox verwendet und es erscheint nur jener Text in der Box, der als zweites Argument übergeben wurde. Damit lässt sich ein individueller Text als Bestätigungsanforderung festlegen.

Wird die Function *Remove-XmlNodeById* innerhalb einer anderen Function mit einem *CmdletBinding*-Attribut und einem „ShouldSupportsProcess=$true“ aufgerufen und wird bei dieser Function der *Confirm*-Parameter gesetzt, wird diese Einstellung an die Function *Remove-XmlNodeById* weitergereicht, so dass vor dem Entfernen des Knotens eine Bestätigungsanforderung erscheint.

Zum Abschluss soll es um eine Variante gehen, durch die der Autor einer Function oder eines Skripts steuern kann, dass auch dann eine Bestätigungsanforderung erscheint, wenn der *Confirm*-Parameter nicht gesetzt wurde. Dafür kommt mit der *ConfirmImpact*-Eigenschaft eine weitere Eigenschaft des *[CmdletBinding]*-Attributs ins Spiel.

Die Eigenschaft *ConfirmImpact* legt fest, wann vor der Ausführung von bestimmten Commands eine Bestätigungsanforderung erscheint, wenn der *Confirm*-Parameter des Commands nicht gesetzt wird. Ansonsten erscheint bekanntlich immer eine solche Anforderung. Für den Parameter kommen die Werte „High“, „Medium“ und „Low“ in Frage. Der Standardwert von *ConfirmImpact* ist „Medium“. Dieser Wert wird immer in Bezug auf den Wert der globalen Variablen *ConfirmPreference* betrachtet, die ebenfalls nur diese drei Werte annehmen kann. Der Standardwert dieser Variablen ist „High“.

Es gilt folgende Regel: Ist der Wert von *ConfirmImpact* bei einem Command gleich oder höher dem Wert in *ConfirmPreference*, wird eine Bestätigungsanforderung ausgegeben; auch dann, und darauf kommt es an, wenn der *Confirm*-Parameter nicht gesetzt wird.

Es liegt am Autor einer Function oder eines Skripts zu entscheiden, wie schwerwiegend die Änderungen sind, die von der Function oder dem Skript vorgenommen werden. Sind es wichtige Änderungen, die einer Bestätigung bedürfen, setzt der Autor den Wert von ConfirmImpact auf „High“. In diesem Fall erscheint immer eine Bestätigungsanforderung (sofern sie nicht mit einem „-confirm:$false“ unterdrückt wird). Sind die Auswirkungen der Function oder des Skripts weniger wichtig, wird der Wert auf „Medium“ belassen (in diesem Fall fällt *ConfirmImpact* einfach weg) oder auf „Low“ gesetzt. In diesem Fall erscheint nur dann eine Bestätigungsanforderung, wenn der Wert in *ConfirmPreference* ebenfalls auf „Low“ gesetzt wird.

Beispiel

Die nächste Version der Function *Remove-XmlNodeById* verwendet die *ConfirmImpact*-Eigenschaft mit dem Wert „High“. In diesem Fall erscheint beim Aufruf der Function auch dann eine Bestätigungsanforderung, wenn der *Confirm*-Parameter nicht gesetzt wird.

```
function Remove-XmlNodeById
{
  # Standardwert von ConfirmImpact ist Medium
  [CmdletBinding(SupportsShouldProcess=$true, ConfirmImpact="High")]
  param([String]$Xml, [String]$NodeId)
  Add-Type -AssemblyName System.Xml.Linq
  $Root = [System.Xml.Linq.XDocument]::Parse($Xml)
  # Es kommt hier auf die Groß-/Kleinschreibung an!
  $Node = $Root.Descendants("book") | Where { $_.Attribute("id").Value -
eq $NodeId }
  # Das Entfernen soll auf Wunsch eine Bestätigungsanforderung ausgeben
  if ($PSCmdlet.ShouldProcess($Node, "Knoten mit Id $Id entfernen?"))
  {
    $Node.Remove()
  }
  $Root.ToString()
}
```

Wird der Wert von *ImpactLevel* auf „Medium“ oder „Low“ gesetzt, erscheint ohne *Confirm*-Parameter keine Bestätigungsanforderung, da der Standardwert der *ConfirmPreference*-Variablen „High“ ist. Wird der Wert der Variablen auf „Medium“ oder „Low“ gesetzt, erscheint wieder eine Anforderung. Damit wird deutlich, wie sich über diese Variablen steuern lässt, dass Commands mit einem niedrigeren „Impact Level“ eine Bestätigungsanforderung anzeigen.

► **Hinweis** Es ist interessant, dass die Werte „Low“, „Medium“ und „High“ nicht voll ausgeschrieben werden müssen.

4.7.4 Eine Bestätigungsanforderung unabhängig von Confirm & Co

Soll eine Function oder ein Skript immer eine Bestätigungsanforderung anzeigen, unabhängig von *ImpactLevel* und *ConfirmPreference*, hat die Variable *PSCmdlet* auch dafür etwas zu bieten. Es ist die Methode *ShouldContinue*. Ihr werden beim Aufruf der Text der Bestätigung und die Überschrift übergeben. Die Variable *PSCmdlet* steht nur zur Verfügung, wenn das *CmdletBinding*-Attribut verwendet wird.

Beispiel

Die Variante der Function *Remove-XmlNodeById*, die im Folgenden vorgestellt wird, fordert vor dem Entfernen des Knotens immer eine Bestätigungsanforderung an, die sich nicht unterdrücken lässt.

```
function Remove-XmlNodeById
{
  # CmdletBinding ist erforderlich, damit es PSCmdlet gibt
  [CmdletBinding()]
  param([String]$Xml, [String]$NodeId)
  Add-Type -AssemblyName System.Xml.Linq
  $Root = [System.Xml.Linq.XDocument]::Parse($Xml)
  # Es kommt hier auf die Groß-/Kleinschreibung an!
  $Node = $Root.Descendants("book") | Where { $_.Attribute("id").Value -
eq $NodeId }
  # Das Entfernen soll immer bestätigt werden müssen
 if ($PSCmdlet.ShouldContinue("Soll der Knoten mit der ID $ID gelöscht
werden?", "Bitte bestätigen" ))
  {
    $Node.Remove()
  }
  $Root.ToString()
}
```

► **Hinweis** Die Variable *PSCmdlet* steht nur dann zur Verfügung, wenn das *[CmdletBinding]*-Attribut verwendet wird.

4.8 Parameter-Validierung

Die PowerShell bietet insgesamt über ein Dutzend Attribute für die Parametervalidierung. Beispiele sind *ValidateRange*, *ValidateScript* und *ValidateSet*, die in diesem Abschnitt anhand von Beispielen vorgestellt werden.

- **Tipp** In der Hilfe sind alle Validierungsattribute unter „about_functions_advanced_parameters“ mit Beispielen beschrieben.

4.8.1 Einen Bereich validieren mit ValidateRange

Soll einem Parameter eine Zahl in einem bestimmten Bereich zugewiesen werden, ist dafür das *ValidateRange*-Attribut zuständig.

Die folgende Validierung stellt sicher, dass sich der Wert für den Parameter *Level* im Bereich −10 bis 10 befindet:

```
param([ValidateRange(-10, 10)][Int]$Level)
```

4.8.2 Eine Auswahlmenge validieren mit ValidateSet

Sollen für einen Parameter vom *String* nur bestimmte Werte erlaubt sein, kommt dafür das *ValidateSet*-Attribut in Frage.

Beispiel

Die folgende Validierung stellt sicher, dass der Wert für den Parameter *Level* nur die Werte „Low“, „Medium“ und „High“ annehmen kann. Diese Werte werden bei der PowerShell ISE auch in eine Auswahlliste angeboten, was sehr praktisch ist.

```
param([ValidateSet("High","Medium","Low")][String]$Level)
```

4.8.3 Ein beliebiges Kriterium validieren per ValidateScript

Das flexibelste Validierungsattribut ist das *ValidateScript*-Attribut, denn hier wird die Validierung über einen Scriptblock erledigt. Der zu validierende Wert wird dabei über die Variable *$_* angesprochen.

Beispiel

Die folgende Validierung stellt sicher, dass der für den Parameter *Path* übergebene Pfad auch existiert.

```
param([ValidateScript({Test-Path -Path $_})][String]$Path)
```

4.8.4 Credential-Parameter implementieren

Eine praktische Eingabehilfe steht automatisch zur Verfügung, wenn ein Parameter explizit vom Typ *[PSCredential]* deklariert wird. Wird für den Parameter beim Aufruf der Function ein Benutzername übergeben, wird das Kennwort automatisch abgefragt und das resultierende *PSCredential*-Objekt der Parametervariablen zugewiesen.

Beispiel

Das folgende Beispiel zeigt die Function-Definition mit einem *PSCredential*-Parameter.

```
function Login-Server
{
  [CmdletBinding()]
  param([PSCredential]$Credential)

}
```

Wird beim Aufruf für den *Credential*-Parameter nur ein String übergeben, erscheint die Anmeldedialogbox für die Eingabe des Kennworts.

4.8.5 Eigene Parameterattribute definieren

Wer mit der Auswahl der vorhandenen Validierungsattribute aus irgendeinem Grund nicht zufrieden ist, kann mit wenig Aufwand eigene Attribute definieren, die nach eigenen Kriterien einen Parameter validieren. Einen praktischen Grund dürfte es nicht geben, da sich per *ValidateScript*-Attribut alles validieren lässt, aber eventuell gibt es jemanden in der weltweiten PowerShell-Community, der genau das tun möchte. Ein kleiner Vorteil eines benutzerdefinierten Attributs ist, dass sich sein Vorhandensein, zum Beispiel im Rahmen der Skriptanalyse über den *PSScriptAnalyzer,* gezielt abfragen lässt. Dürfen aufgrund einer unternehmensweiten Vorgabe die Werte für ein Benutzerkonto keine Umlaute enthalten, ließe sich das per *ValidateScript* natürlich abfragen. Es wäre aber etwas aufwändiger, im Rahmen einer Analyse abzufragen, ob in einem Skript diese Validierungsregel auch angewendet wird. Wird für die Validierung daher ein eigenes Attribut verwendet, wird eine solche Abfrage sehr einfach.

Nach der langen Vorrede hier ein kleines Beispiel. In dem Beispiel wird eine neue Attributklasse mit dem Namen „ValidateWeekendAttribute“ definiert. Sie soll überprüfen, ob ein Datum auf ein Wochenende fällt. Eine benutzerdefinierte Attributklasse muss sich von der Klasse *ValidateArgumentsAttribute* im Namespace *System.Management.Automation* ableiten. Dank des *class*-Befehls ist die Definition einer solchen Klasse sehr einfach.

Beispiel

Das folgende Beispiel zeigt eine Klasse mit dem Namen „ValidateWeekendAttribute“, die bei einem Datumsparameter überprüft, ob das Datum generell gültig ist, und ob es ein Samstag oder Sonntag ist. Ist Letzteres der Fall, wird eine Fehlermeldung ausgegeben.

```
<#
 .Synopsis
 Benutzerdefiniertes Parameter-Attribut
#>

class ValidateWeekendAttribute :
System.Management.Automation.ValidateArgumentsAttribute
{

  [void]Validate([Object]$Value,
[System.Management.Automation.EngineIntrinsics]$EngineIntrinsics)
  {
    if (-not (Get-Date -Date $Value -ErrorAction Ignore))
    {
        throw "Parameter besitzt ein ungültiges Datum"
    }

    [DayOfWeek]$w = (Get-Date -Date $Value).DayOfWeek
    if ($w -eq "Saturday" -or $w -eq "Sunday")
    {
        throw "Datum darf weder Samstag noch Sonntag sein"
    }
  }
}
```

Anwendet wird das neue Attribut genau wie die eingebauten Attribute.

Beispiel

Das folgende Beispiel wendet beim Parameter *Date* eine Validierung mit dem *ValidateWeekendAttribute*-Attribut an.

```
function Test-Date
{
  param([ValidateWeekendAttribute()][String]$Date)
```

4.9 Dynamische Parameter

Ein dynamischer Parameter ist ein Parameter, der erst nachträglich in Abhängigkeit einer Bedingung zu einer Function oder einem Script hinzugefügt wird. Die PowerShell macht von dynamischen Parametern exzessiv Gebrauch, indem zum Beispiel bei den verschiedenen Item-Cmdlets bestimmte Parameter in Abhängigkeit des über den Pfad ausgewählten Providers zur Verfügung stehen. Wird zum Beispiel über den *Path*-Parameter des *Get-Item*-Cmdlets ein Dateisystempfad ausgewählt, wird unter anderem der Parameter *Stream* angeboten. Wird über das Cmdlet ein Registry-Pfad ausgewählt, gibt es den Parameter nicht.

Dynamische Parameter lassen sich mit dem *DynamicParam*-Befehlswort zu einer Function hinzufügen, wenngleich es dafür in der Praxis nur selten eine Notwendigkeit geben dürfte. Die Function muss dazu zwei Voraussetzungen erfüllen.

1. Die Function muss eine Pipeline-Function sein und einen *end*-Block enthalten.
2. Das *CmdletBinding*-Attribut muss verwendet werden.

Das Hinzufügen eines dynamischen Parameters besteht aus mehreren Schritten. Die Function beginnt mit dem *DynamicParam*-Befehlswort, das unmittelbar auf den *param*-Befehl folgt, und das einen Scriptblock einleitet, in dem der dynamische Parameter hinzugefügt wird. Als erstes wird in dem Scriptblock ein Parameterattribut definiert. Hier kann der Parameter einem Parameterset zugeordnet werden. Im zweiten Schritt wird das Parameterattribut zu einer Collection hinzugefügt, in der alle Attribute gesammelt werden. Die Collection ist eine generische Collection und muss zuvor angelegt werden. Im dritten Schritt wird der dynamische Parameter mit der Parameterattribut-Collection angelegt. Im vierten und letzten Schritt wird mit dem *RuntimeDefinedParameterDictionary* ein weiteres Objekt angelegt, dem der dynamische Parameter hinzugefügt wird. Dieses „Nachschlagewerk", das alle dynamischen Parameter umfasst, wird in die Pipeline gelegt.

Beispiel

Das folgende Beispiel definiert eine Function *Get-TextData* mit einem dynamischen Parameter *AsXml*. Dieser kann nur dann verwendet werden, wenn der Pfad, der dem *Path*-Parameter zugeordnet wird, die Erweiterung „.xml" besitzt.

```
function Get-TextData
{
  [CmdletBinding()]
  param([String]$Path)
  DynamicParam
  {
    if ([System.IO.Path]::GetExtension($Path) -eq".xml")
    {
      $Attributes = New-Object -TypeName
System.Management.Automation.ParameterAttribute
      $Attributes.ParameterSetName = "__AllParameterSets"
      $Attributes.Mandatory = $false
      $AttributeCollection = New-Object -TypeName
System.Collections.ObjectModel.Collection[Attribute]
      $AttributeCollection.Add($Attributes)
      $SwitchType = [System.Management.Automation.SwitchParameter]
      $DynParameter = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter -ArgumentList
"AsXml", $SwitchType, $AttributeCollection
      $DynParameterDic = New-Object -Type
Management.Automation.RuntimeDefinedParameterDictionary
      $DynParameterDic.Add("AsXml", $DynParameter)
      $DynParameterDic
    }
  }
  end {
   if ($PSBoundParameters.AsXml -ne $null) {
      [Xml](Get-Content -Path $Path)
   }
   else {
      Get-Content -Path $Path
   }
  }
}
```

4.10 Zusammenfassung

Advanced Functions unterscheiden sich von regulären Functions durch erweiterte Parameter-Definitionen und das (optionale) *CmdletBinding*-Attribut. Das *CmdletBinding*-Attribut fügt die allgemeinen Parameter hinzu und aktiviert dieselben Regeln für die Parameterwert-Übergabe wie bei einem Cmdlet. Über die Eigenschaften des Attributs werden unter anderem die Parameter *Confirm* und *WhatIf* hinzugefügt. Durch ein Setzen des „Impact Level“ auf „High“ wird erreicht, dass für jede Function und jedes Skript, das mit einem *Confirm*-Parameter aufgerufen wird, und dessen Impact Level „Medium“ oder „Low“ ist, der *Confirm*-Parameter wirksam wird.

Aus Functions und Skripte werden Module

5

Zusammenfassung

Damit sich PowerShell-Funktionalitäten, die als Functions und Skripte vorliegen, flexibel im Unternehmensnetzwerk verteilen lassen, müssen sie als Module vorliegen. In diesem Kapitel wird gezeigt, wie aus einem Skript über das Zusammenfassen von PowerShell-Befehlen in Functions Module werden, die entweder per Copy-Kommando oder per PowerShell-Paketmanager komfortabel auf einem Arbeitsplatz hinzugefügt werden.

5.1 Module und Modultypen

Bei der PowerShell spielen Module eine zentrale Rolle. Module wurden mit der Version 2.0 eingeführt. Mit der Version 3.0 wurde die PowerShell selber auf Module umgestellt. Jedes Cmdlet der PowerShell gehört daher zu einem Modul. Ein

```
Get-Command -CommandType Cmdlet | Group-Object Module | Sort-Object Count
-Descending
```

gruppiert die vorhandenen Cmdlets nach den Modulen, zu denen sie gehören und gibt die Namen der Module sortiert nach der Anzahl an Cmdlets pro Modul aus.

Ein Modul ist ein Verzeichnis, dessen Inhalt in Gestalt von Dateien beim Laden des Moduls in die PowerShell-Sitzung geladen wird. Die Dateien sind Modulskriptdateien (Erweiterung *.psm1*), Modulmanifestdateien (Erweiterung *.psd1*), reguläre Skriptdateien, Assembly-Dateien (Erweiterung *.dll*), sowie Typen- und Formatdefinitionsdateien (Erweiterung *.ps1xml*) und in der Regel auch Hilfedateien. Voraussetzung für ein Modulverzeichnis ist lediglich eine Psm1- oder Psd1-Datei oder beides. Ein Modul-Verzeichnis kann überall

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_5

angelegt werden. In der Regel wird es aber ein Verzeichnis sein, dessen Pfad sich in der Umgebungsvariablen *PSModulePath* befindet. Diese Verzeichnisse werden automatisch durchsucht, so dass Module, die sich in einem der *PsModulePath*-Verzeichnisse befinden, implizit geladen werden. Ansonsten muss ein Modul explizit per *Import-Module*-Cmdlet mit dem vollständigen Pfad des Modulverzeichnisses geladen werden.

Es werden vier Modultypen unterschieden:

1. Skriptmodule,
2. Manifestmodule,
3. binäre Module und
4. dynamische Module.

5.1.1 Skriptmodule

Ein Skriptmodul ist ein Verzeichnis mit einer Psm1-Datei. Diese Modulskriptdatei unterscheidet sich von einer Ps1-Datei nicht durch ihren Inhalt, sondern durch den Umstand, dass sie nicht direkt ausgeführt werden kann. Sie enthält in der Regel die Function-Definitionen, die über das Modul geladen werden. Ein weiterer kleiner Unterschied: In einer Psm1-Datei kann per *Export-ModuleMember* festgelegt werden, welche Functions, Variablen, Cmdlets und Aliase über das Modul zur Verfügung gestellt werden sollen. Das ist für Aliase wichtig, da sie ansonsten nicht sichtbar sind.

5.1.2 Manifestmodule

Ein Manifestmodul besteht aus einer Psd1-Datei. Diese Datei enthält eine Beschreibung des Moduls in Gestalt einer Hashtable. Neben einem Namen und einer Beschreibung gehört dazu auch die Versionsnummer des Moduls. Manifestmodule können daher in verschiedenen Versionen parallel vorliegen, so dass per *Import-Module*-Cmdlet gezielt eine bestimmte Version geladen werden kann. Bei einem Skriptmodul ist das nicht möglich. Wenn die Versionierung des Moduls eine Rolle spielen soll, muss ein Manifestmodul angelegt werden.

5.1.3 Binärmodule

Ein Binärmodul entsteht durch direktes Laden einer Assembly-Datei mit Cmdlet-Definitionen per *Import-Module*-Cmdlet. Danach stehen die Cmdlets im Rahmen der PowerShell-Sitzung zur Verfügung. Eine Manifest- oder eine Psm1-Datei gibt es in diesem Fall nicht.

Beispiel

Der folgende Befehl lädt eine Assembly-Datei mit einer Reihe von Cmdlet-Definitionen.

```
Import-Module -Name .\PSInfoCmdlet.dll
```

Ein *Get-Module* listet das geladene Modul mit seinen Cmdlets auf.

5.1.4 Dynamische Module

Ein dynamisches Modul ist ein Modultyp, der im PowerShell-Alltag nur selten eine Rolle spielt, auch wenn sich seine Möglichkeiten zunächst nach einer guten Idee anhören: Über das *New-Module*-Cmdlet werden Cmdlets und Functions zu einem Modul zusammengefasst. Alternativ lassen sich diese auch über ein Objekt und dessen Members ansprechen. Im Unterschied zu den anderen Modultypen wird ein dynamisches Modul nicht in die globale Modultabelle aufgenommen und kann daher nicht per *Get-Module*-Cmdlet abgefragt werden.

Beispiel

Das folgende Beispiel legt per *New-Module*-Cmdlet ein dynamisches Modul an, das die Function eines Scriptblocks als Members zur Verfügung stellt.

```
$SB = {
  function f1
  {
    "this is f1..."
  }

  function f2
  {
    "this is f2..."
  }
}

New-Module -Name MeineFuncs -ScriptBlock $SB -AsCustomObject
```

Der *AsCustomObject*-Parameter soll dafür sorgen, dass ein Objekt resultiert, über dessen Members die beiden Functions aufgerufen werden können. Doch beim Aufruf von *New-Module* passiert nichts. Der Grund: Das Objekt steht erst dann zur Verfügung, wenn es einer Variablen zugewiesen wird.

Beispiel

Der folgende Aufruf von *New-Module* macht aus einem Scriptblock mit Functions ein Objekt.

```
$M = New-Module -Name MeineFuncs  -ScriptBlock $SB -AsCustomObject
```

Soll aus dem dynamischen Modul ein richtiges Modul werden, muss auf *New-Module* ein *Import-Module* folgen. Das Modul steht dann im Rahmen der PowerShell-Sitzung zur Verfügung.

Eine praktische Anwendung für dynamische Module ergibt sich beim Importieren eines Moduls aus einer Remote-Session. Dies ist beim Exchange-Server aktuell die gängige Praxis, damit die Exchange-Server-Cmdlets, die sich physisch auf dem Exchange-Server befinden, im Rahmen einer Remote-Session auf dem Client über Proxy-Functions ausgeführt werden können.

5.1.5 Weitere Modulverzeichnisse hinzufügen

Sollen weitere Verzeichnisse als Modulverzeichnisse bei der Ausführung eines Commands, das sich in einem noch nicht geladenen Modul befindet, berücksichtigt werden, muss der Verzeichnispfad lediglich zur Umgebungsvariablen *PSModulePath* hinzugefügt werden.

```
$env:PSModulePath += "F:\EigeneModule"
```

Dies erspart, dass das Modul explizit per *Import-Module* geladen werden muss. Auch wenn es theoretisch möglich ist, sollte ein UNC-Pfad nicht hinzugefügt werden, da die PowerShell in regelmäßigen Abständen die dort abgelegten Verzeichnisse durchsucht, was zu kleinen Verzögerungen führen kann. Soll ein Modul zum Beispiel im Unternehmensnetzwerk zur Verfügung gestellt werden, ist das Einrichten eines Repository (mehr dazu im weiteren Verlauf dieses Kapitels) die bessere, weil flexiblere Option.

5.2 Manifestmodule im Detail

Ein Manifestmodul ist ein Modul, dessen Inhalt durch eine Manifestdatei beschrieben wird und das damit über Metadaten verfügt. Die Manifestdatei trägt die Erweiterung *.psd1* und enthält eine Reihe von Einstellungen in Gestalt einer Hashtable. Am einfachsten wird die Datei über das *New-ModuleManifest*-Cmdlet angelegt. Es wird mit dem Pfad einer Psd1-Datei aufgerufen (die Erweiterung muss angegeben werden). Die Manifestdatei enthält alle möglichen Einträge versehen mit Kommentaren, von denen viele auskommentiert sind, da sie nur selten eine Rolle spielen. Lassen Sie sich daher von der Fülle an Text nicht irritieren, eine Manifestdatei kann sehr einfach sein.

▶ **Tipp** Das *Test-ModuleManifest*-Cmdlet gibt an, ob eine Manifestdatei bezüglich ihres Aufbaus fehlerfrei ist. Beim Laden eines Moduls werden ebenfalls fehlerhafte Einträge angezeigt.

Tab. 5.1 fasst die wichtigsten Einstellungen einer Manifestdatei zusammen. Einträge wie *Author*, *Description* oder *CompanyName* sollten selbsterklärend sein. Andere Einträge

Tab. 5.1 Die Einstellungen einer Modulmanifestdatei

Eintrag	Bedeutung
RootModule	Legt das Stammmodul fest. Dies kann zum Beispiel eine Psm1-Datei sein oder eine Psd1-Datei in einem anderen Verzeichnis. Der Eintrag kann auch entfallen und das oder die Module über *NestedModules* festgelegt werden. Auch wenn es in der Praxis oft der Fall ist, muss der Name der Psm1-Datei nicht mit dem Namen des Modulverzeichnisses übereinstimmen.
ModuleVersion	Legt die Versionsnummer des Moduls fest; sollte in der Form 1.0.0.0 angegeben werden.
RequiredModules	Legt das oder die Modul(e) fest, die vor dem Laden des Moduls geladen werden müssen. Ersetzt den Eintrag *ModulesToProcess*, der nicht mehr verwendet werden sollte. Auch wenn nur eine Datei angegeben wird, sollte diese in der vorgegebenen Array-Schreibweise angegeben werden.
RequiredAssemblies	Legt den oder die Assembly-Datei(en) fest, die vor dem Laden des Moduls geladen werden sollen. Spielt nur dann eine Rolle, wenn eine Function als Teil eines Modulskripts die Assembly ansprechen soll, ohne dass das *Add-Type*-Cmdlet ausgeführt werden soll. Ansonsten spielt es in der Praxis keine Rolle. Auch für diese Einstellung gilt, dass wenn nur eine Datei angegeben wird, diese trotzdem in der Array-Schreibweise angegeben werden sollte.
ScriptsToProcess	Sollten mit dem Laden des Moduls ein oder mehrere Skripte ausgeführt werden, werden ihre Namen hier angegeben. Spielt in der Praxis keine Rolle.
TypesToProcess	Hier werden die Namen von XML-Dateien mit der Erweiterung .types.ps1xml angegeben, in denen Typendefinitionen enthalten sind. Wird in der Praxis nur selten benötigt.
FormatsToProcess	Hier werden die Namen von XML-Dateien mit der Erweiterung *.formats.ps1xml* angegeben, in denen Formatdefinitionen für Typen enthalten sind, die in dem Modul eine Rolle spielen. Wird in der Praxis nur selten benötigt.
NestedModules	Wichtiger Eintrag, da hier weitere Module angegeben werden, die mit dem Modul importiert werden. Die Module werden über ihren Namen oder ihren Verzeichnispfad angegeben.
FunctionsToExport	Falls das Modul nicht jede Function exportieren soll, müssen die exportierten Functions hier einzeln aufgeführt werden.
CmdletsToExport	Falls das Modul nicht jedes Cmdlet exportieren soll, müssen die exportierten Functions hier einzeln aufgeführt werden.
VariablesToExport	Falls das Modul nicht jede Variable exportieren soll, müssen die exportierten Functions hier einzeln aufgeführt werden.
AliasesToExport	Falls das Modul nicht jeden Alias exportieren soll, müssen die exportierten Functions hier einzeln aufgeführt werden.
ModuleList	Hier können alle Module aufgeführt werden, die bei dem Modul eine Rolle spielen. Spielt in der Praxis keine Rolle.
FileList	Hier können alle Dateien aufgeführt werden, die bei dem Modul eine Rolle spielen. Spielt in der Praxis keine Rolle.

(*Fortsetzung*)

Tab. 5.1 (Fortsetzung)

Eintrag	Bedeutung
PrivateData	In diesem Bereich werden Daten in Gestalt einer Hashtable abgelegt, die das Modul begleiten.
DefaultCommandPrefix	Mit diesem Eintrag wird ein Präfix festgelegt, der jedem Hauptwortanteil eines Command-Namens vorangestellt wird. Damit werden die Namen der Commands personalisiert und es besteht keine Verwechslungsgefahr mit anderen Commands. Praktisch, da das Modul ansonsten per *Import-Module* explizit importiert werden müsste.

wie *PowerShellVersion* oder *ModuleList* erscheinen eventuell wichtig, spielen aber in der Praxis in der Regel keine Rolle. Jedem Eintrag geht eine Kommentarzeile voraus, der seine Bedeutung erklärt. Der einzige obligatorische Eintrag einer Modulmanifestdatei ist entweder *RootModule* oder *NestedModules*, da darüber zum Beispiel auf eine Psm1-Datei verwiesen wird, ohne die das Modul ansonsten mangels Inhalt keine echte Funktion erfüllen würde. Soll das Modul in einem Repository veröffentlicht werden, sind die Einträge *ModuleVersion*, *Author* und *Description* obligatorisch. Die Versionsnummer muss in der allgemeinen Form „1.0.0.0“ angegeben werden.

- **Tipp** Das *PowerShellGet*-Modul enthält die Function *Update-ModuleManifest*, die einzelne Einträge einer Manifestdatei aktualisiert.

- **Hinweis** Wird per *RootModule* eine Psm1-Datei angegeben, wird das Modul offiziell als Scriptmodul geführt. Ist dies nicht gewünscht, muss die Psm1-Datei stattdessen über den Eintrag *NestedModule* festgelegt werden.

5.3 Verschachtelte Module

Nicht immer soll die gesamte Funktionalität in einem Modul enthalten sein. Es existieren mehrere Module mit jeweils einem Funktionsschwerpunkt. Diese Module sollen aber nicht nur einzeln, sondern auch als Teil eines anderen Moduls geladen werden. Ein verschachteltes Modul umfasst mehrere Module, die über den Eintrag *NestedModules* in der Manifestdatei festgelegt werden, entweder über ihren Namen oder den Verzeichnispfad des Modulverzeichnisses.

5.4 Mehrere Versionen eines Moduls verwenden

Seit der Version 5.0 ist es problemlos möglich, mehrere Versionen eines Moduls in einem Modulverzeichnis abzulegen. Jede Version liegt in ihrem eigenen Unterverzeichnis vor, dessen Name der Versionsnummer entspricht. Wird ein Modul über das

Install-Module-Cmdlet aus einem Repository hinzugefügt, werden die Dateien des Moduls automatisch in einem Unterverzeichnis abgelegt, das der Versionsnummer entspricht. Bei Modulen, die „zu Fuß" angelegt werden, muss das Verzeichnis selber angelegt werden, wenn das Modul in mehreren Versionen verfügbar sein soll. Über die Parameter *RequiredVersion*, *MinimumVersion* und *MaximumVersion* werden gezielt bestimmte Versionen des Moduls geladen. Das *Get-Module*-Cmdlet gibt seit der Version 5.0 die Versionsnummer zu einem Modul an.

5.5 Internationale Module – Zeichenketten in Psd1-Dateien auslagern

Sollen die Zeichenketten der Psm1- und Ps1-Dateien eines Moduls in einer separaten Datei ausgelagert werden, geschieht dies im Allgemeinen über eine Psd1-Datei, in der die Zeichenketten als Teil eines Here-Strings nach dem Schema „Schlüssel = Zeichenkette" abgelegt sind. Der Here-String wird in der Psd1-Datei über das *ConvertFrom-StringData*-Cmdlet in eine Hashtable konvertiert. Über das *Import-LocalizedData*-Cmdlet wird die Psd1-Datei im Rahmen eines Skripts geladen, so dass die Hashtable anschließend über die bei *Import-LocalizedData* angegebene Variable zur Verfügung steht.

Soll die Psd1-Datei automatisch anhand der aktuell eingestellten „Länderkennung" (Kulturinfo) geladen werden, muss ihr Name dem Namen des Modulverzeichnisses entsprechen und sie muss sich in einem Unterverzeichnis befinden, dessen Namen der jeweiligen Kulturbezeichnung entspricht, zum Beispiel „de" für Deutsch oder „en" für Englisch usw.

Möchte man erreichen, dass beim Laden eines Moduls die in den einzelnen Ps1- und Psm1-Dateien verwendeten Zeichenketten automatisch aus der zur aktuell eingestellten Landessprache passenden Psd1-Datei geladen werden, ist dies mit drei einfachen Schritten machbar:

1. Pro Landessprache wird im Modulverzeichnis ein Unterverzeichnis mit dem Kürzel der Landessprache angelegt, zum Beispiel „de" für Deutschland, „en" für Englisch oder „fr" für Frankreich. Theoretisch kann auch der Name der Region einbezogen werden, also „de-DE" oder „en-GB", aber in der Praxis ist das nicht erforderlich.
2. Jedes Unterverzeichnis enthält eine Textdatei mit dem Namen der Skriptdatei und der Erweiterung *.psd1*.
3. Jede dieser Textdateien enthält die Zeichenketten in Gestalt einer Liste von „Name = Zeichenkette"-Einträge, die im Rahmen der Psd1-Datei über das *ConvertFrom-StringData*-Cmdlet als Hashtable zur Verfügung gestellt wird.

Beispiel

Das folgende Beispiel geht von einem Modul mit dem Namen „PSInfo" aus, das eine Psm1-Datei mit dem Namen „PsInfo" enthält, deren Meldungen in den Sprachen Deutsch, Englisch und Französisch in jeweils separaten Psd1-Dateien vorliegen sollen.

Die Psd1-Datei trägt daher immer den Namen „PSInfo.psd1“ und befindet sich mit unterschiedlichen Inhalten in den Unterverzeichnissen „de“, „en“ und „fr“ in dem Verzeichnis, in dem sich auch die Psm1-Datei befindet.

Eine Psd1-Datei im Unterverzeichnis „de“ ist wie folgt aufgebaut:

```
ConvertFrom-StringData -StringData @'
MorningGreet = Guten Morgen
DayGreet = Guten Tag
EveningGreet = Guten Abend
StandardGreet = Alles klar?
'@
```

Entsprechend der Aufbau der Psd1-Datei im Unterverzeichnis „en“:

```
ConvertFrom-StringData -StringData @'
MorningGreet = Good morning
DayGreet = Good day
EveningGreet = Good evening
StandardGreet = What`s app?
'@
```

Es fällt auf, dass bei der zugewiesenen Zeichenkette keine Anführungszeichen oder Apostrophe im Spiel sind.

Fehlt noch die Psm1-Datei. Sie ist wie folgt aufgebaut:

```
<#
 .Synopsis
 Ausgabe einer internationalen Begrüßung
#>

Import-LocalizedData -BindingVariable Greetings

function Write-Hello
{
  switch ((Get-Date).Hour)
  {
    { $_ -ge 5 -and $_ -le 11 } { $Greetings.MorningGreet }
    { $_ -ge 12 -and $_ -le 17 } { $Greetings.DayGreet }
    { $_ -ge 18 -and $_ -le 23 } { $Greetings.EveningGreet }
    default { $Greetings.StandardGreet }
  }
}
```

An der Ps1m-Datei fällt natürlich sofort auf, dass die Psd1-Datei nirgendwo referenziert wird. Das ist nicht erforderlich, da die Zuordnung über das *Import-LocalizedData*-Cmdlet anhand der Benennung der Unterverzeichnisse automatisch geschieht. Die eingelesenen Zeichenketten werden über eine Variable angesprochen, in diesem Beispiel ist es die Variable *Greetings*.

Mehr ist nicht zu tun. Wird das Modul unter einem Windows mit englischer Spracheinstellung geladen, erscheint die tageszeitabhängige Begrüßung automatisch auf Englisch.

5.5.1 Die Rolle der Kultur und das CultureInfo-Objekt

Leider ist das Umschalten auf eine andere Sprache bei Windows etwas aufwändiger und bei installiertem Sprachpaket mit einer Neuanmeldung verbunden. Für das Testen der unterschiedlichen Sprachen gibt es beim *Import-LocalizedData*-Cmdlet den *UICulture*-Parameter. Die über diesen Parameter angegebene Kultur, zum Beispiel „en“, bestimmt, welche Psd1-Datei geladen wird.

Das führt unweigerlich zur Frage: „Was bitte versteht Microsoft unter einer Kultur?“ Dahinter steckt der mit der .NET-Laufzeit eingeführte Begriff der „Culture“, der den bis dahin unter Windows verwendeten Begriff der „Locale“ ablöst beziehungsweise auf eine erweiterte und vor allem vereinheitlichte Grundlage stellt. Bei der .NET-Laufzeit und damit auch bei der PowerShell steht der Begriff „Culture“ für die Summe aller landesspezifischen Einstellungen. Die aktuelle Kultur wird am einfachsten über das *Get-Culture*-Cmdlet abgefragt. Wenn die Rückgabe zunächst unspektakulär erscheint, ändert sich das, sobald ein „Select-Object *“ angehängt wird. Die folgende Abfrage gibt zum Beispiel alle Einstellungen zurück, die die Formatierung von Datums- und Zeitwerten betreffen:

```
Get-Culture | Select-Object –ExpandProperty DatetimeFormat
```

Es gibt weitere Alternativen zur Abfrage der aktuellen Kultureinstellung: Die Variable Host, die die Host-Anwendung als Ganzes repräsentiert, mit den beiden Eigenschaften *Culture* und *UICulture*. Und für Scripting-affine Admins:

```
[System.Threading.Thread]::CurrentThread.CurrentCulture
```

beziehungsweise

```
[System.Threading.Thread]::CurrentThread.CurrentUICulture
```

Die Unterscheidung zwischen *Culture* und *UICulture* spielt bei der PowerShell keine Rolle. *Culture* ist für die Einstellungen zuständig, *UICulture* würde festlegen, welche externen Ressourcendateien geladen werden. Da es diese aber bei der PowerShell nicht gibt, spielt auch die Einstellung von *UICulture* keine Rolle. Beide Eigenschaften liefern stets denselben Wert.

Alle Kultureinstellungen werden durch ein *CultureInfo*-Objekt im Namespace *System.Globalization* zusammengefasst. Die Eingabe von

```
[CultureInfo]"es-ES"
```

liefert ein *CultureInfo*-Objekt mit allen Einstellungen für die Sprache Spanisch und die Region Spanien. Als Kulturbezeichner kann auch nur die Sprache angegeben werden, zum Beispiel „de“. Gibt es einen Kulturbezeichner nicht, wird daraus ein unbekanntes Gebietsschema oder eine unbekannte Sprache.

Beispiel

Wer sich für Details interessiert, der folgende Befehl liefert alle der insgesamt 203 (!) Kulturbezeichner.

```
[System.Globalization.CultureInfo]::GetCultures("FrameworkCultures")
```

Offiziell gibt es deutlich mehr, nämlich 803, darunter auch „Tadschikisch (Kyrillisch, Tadschikistan)“.

5.5.2 Ändern der aktuellen Kultur

Für das Ändern der aktuellen Kultur gibt es ab Windows Server 2012 und Windows 8.1 das Cmdlet *Set-Culture* aus dem *International*-Modul. Leider besitzt das Cmdlet eine kleine Besonderheit: Es funktioniert nicht. Es funktioniert zumindest nicht so, wie man es erwarten würde, denn der Aufruf ändert nicht die aktuelle Kultureinstellung in einer PowerShell-Sitzung. Da der Grund dafür darin liegt, dass offenbar nach der Ausführung jedes (!) Cmdlets die Kultur wieder auf die Windows-Einstellung zurückgesetzt wird, gibt es seit Jahren einen „Workaround“ für den Fall, dass ein Befehl aus welchen Gründen auch immer unter einer anderen Kultureinstellung ausgeführt werden soll. Dieser besteht darin, dass der Befehl in einem Scriptblock ausgeführt und die aktuelle Kultur unmittelbar vor der Ausführung auf die neue Kultur und unmittelbar danach wieder auf die aktuelle Kultur zurückgesetzt wird. Was eventuell ein wenig (zu) kompliziert klingen mag, ist im Grunde sehr einfach, wenn man sich die involvierten Befehle betrachtet:

```
<#
 .Synopsis
 Setzen der Kultur für die Ausführung eines Scriptblocks
#>
function Using-Culture
{
  param([ScriptBlock]$Scriptblock,
        [CultureInfo]$Culture)
  $oldCulture = [System.Threading.Thread]::CurrentThread.CurrentCulture
  $oldUICulture =
[System.Threading.Thread]::CurrentThread.CurrentUICulture
  try
  {
    [System.Threading.Thread]::CurrentThread.CurrentCulture = $Culture
    [System.Threading.Thread]::CurrentThread.CurrentUICulture = $Culture
    & $Scriptblock
  }
  finally
  {
    [System.Threading.Thread]::CurrentThread.CurrentCulture = $oldCulture
    [System.Threading.Thread]::CurrentThread.CurrentUICulture =
$oldUICulture
  }
}
```

Beispiel

Möchte man zum Beispiel sehen, wie das aktuelle Datum und die aktuelle Uhrzeit auf Finnisch ausgegeben werden, erledigt das der folgende Aufruf.

```
Using-Culture -Scriptblock { Get-Date} -Culture fi
```

Das ist natürlich in erster Linie eine Spielerei. Der Grund dafür, dass die Function *Using-Culture* überhaupt ins Spiel kam, war, dass damit das *PsInfo*-Modul unter den verschiedenen Landeseinstellungen getestet werden kann. Auch das ist natürlich möglich. Der folgende Befehl lädt das Modul mit einer französischen Spracheinstellung:

```
Using-Culture -Scriptblock { Import-Module .\psinfo -force; write-hello}
-Culture fr
```

Der *Force*-Parameter bei *Import-Module* sorgt dafür, dass ein Modul auch dann geladen wird, wenn es bereits geladen ist.

5.6 Praxisteil: Erstellen eines Manifestmoduls

In diesem Praxisteil wird ein Manifestmodul mit dem Namen „PsInfo“ umgesetzt, das neben einer Psd1- und einer Psm1-Datei auch eine Assembly-Datei mit Cmdlet-Definitonen, eine allgemeine Assembly-Datei sowie eine Ps1-Datei mit „Helper-Functions“ umfasst. Auf eine Typen- und Formatdefinitionsdatei wurde verzichtet. Sie finden das Modul als Teil der Beispieldateien für dieses Buch. Falls Sie die Übung direkt umsetzen möchten, lassen Sie die beiden Schritte einfach aus, in denen eine Dll-Datei mit Visual Studio erstellt wird und verwenden Sie für die Psm1- und Ps1-Datei eine einfache Function, die lediglich eine Meldung ausgibt.

1. Im ersten Schritt wird zum Beispiel unter „C:\Program Files\WindowsPowerShell\Modules“ ein Verzeichnis mit dem Namen „PsInfo“ angelegt.
2. Im nächsten Schritt wird in diesem Verzeichnis eine Psm1-Datei mit dem Namen „PsInfoFunctions.psm1“ angelegt. Der Name der Datei soll absichtlich nicht dem Verzeichnisnamen entsprechen, da dies bei einem Manifestmodul keine Voraussetzung ist. Die Ps1-Datei enthält eine Reihe von Function-Definitionen.
3. Im nächsten Schritt wird in dem Modulverzeichnis die Ps1-Datei „PsInfoHelpers.ps1“ angelegt (sie könnte sich auch in einem anderen Verzeichnis befinden). Sie enthält ebenfalls eine Reihe von Function-Definitionen.
4. Im nächsten Schritt wird eine Assembly-Datei *PsInfoLib.dll*, zum Beispiel mit Hilfe von Visual Studio 2015 Community Edition, umgesetzt. Diese Assembly enthält eine Reihe von Methoden. Diese sind nicht PowerShell-spezifisch und sollen von einem PowerShell-Cmdlet aufgerufen werden. Die Dll-Datei wird im Modulverzeichnis abgelegt.

5. Im nächsten Schritt wird eine Assembly-Datei *PsInfoCmdlet.dll,* zum Beispiel mit Hilfe von Visual Studio 2015 Community Edition, umgesetzt. Diese Assembly enthält eine Reihe von Cmdlet-Definitionen, die auf Methoden der Assembly *PsInfoLib.dll* zugreifen. Beide Dll-Dateien müssen sich daher im selben Verzeichnis, in diesem Fall das Modulverzeichnis, befinden.

Jetzt kommt der wichtigste Schritt. Es wird eine Modulmanifestdatei mit dem Namen „PsInfo.psd1“ angelegt, in der alle aufgezählten Dateien untergebracht werden. Die Datei ist wie folgt aufgebaut.

```
@{
  ModuleVersion = '1.0'
  GUID = '053c22df-cf86-4793-b2e2-196334e71425'
  Author = 'Pemo'
  Description = 'Cmdlets für das Abrufen von Systeminformationen'
  PowerShellVersion = '3.0'
  ScriptsToProcess = 'Helpers.ps1'
  NestedModules = @('PsInfo.psm1', 'PSInfoCmdlet.dll')
  FunctionsToExport = '*'
  CmdletsToExport = '*'
}
```

Damit ist das Modulmanifest fertig (Abb. 5.1). Ein

```
Get-Module -Name PsInfo -ListAvailable
```

zeigt das Modul als Manifestmodul an. Es wird automatisch mit dem Ausführen eines Cmdlets oder einer Function geladen, die in der Psm1- oder auch Ps1-Datei enthalten ist.

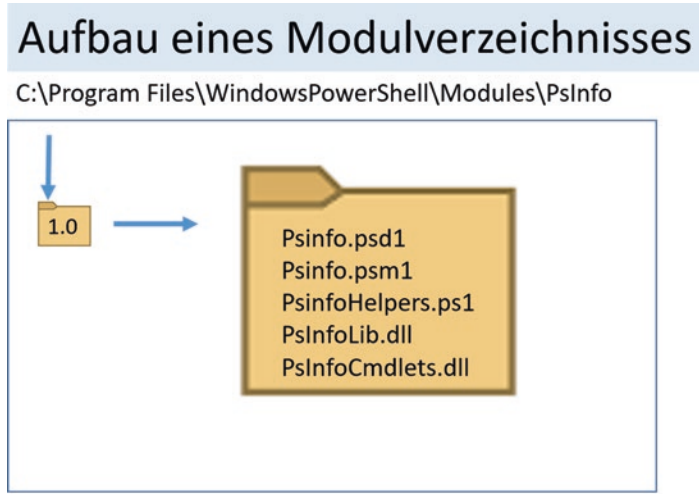

Abb. 5.1 Der Inhalt eines Manifestmodul-Verzeichnisses an einem Beispiel

5.7 Zusammenfassung

Module sind bei der PowerShell der zentrale Mechanismus für das Hinzufügen von Funktionalität. Ein Modul basiert auf einem Verzeichnis, das sich in der Regel in einem der Modulverzeichnisse befindet, deren Pfade in der Umgebungsvariablen *PSModulePath* enthalten sind. Seit der Version 3.0 werden auch die internen Cmdlets der PowerShell als Module geladen. Ein „Get-Module | Remove-Module“ entfernt alle Module bis auf das Kernmodul *Microsoft.PowerShell.Management*. Module werden in der Regel implizit geladen. Das heißt durch die Eingabe eines Commands wird automatisch das Modul geladen, in dem das Command definiert wird (die PowerShell legt dafür intern einen Zwischenspeicher, den „Module Cache“ an). Die wichtigste Modulsorte sind Manifestmodule, da im Rahmen der Manifestdatei unter anderem die Versionsnummer des Moduls angegeben werden kann. Sollen alle PowerShell-Anwender auf einen definierten und getesteten Satz von Modulen zugreifen können, empfiehlt es sich, im Intranet ein eigenes Modul-Repository anzulegen.

PowerShell-Skripte testen mit Pester

6

Zusammenfassung

Je umfangreicher ein Skript wird, desto höher ist die Wahrscheinlichkeit, dass irgendetwas nicht so ausgeführt wird, wie es sich der Autor des Skriptes vorgestellt hat. Das gilt vor allem, wenn es mehrere Autoren gibt, die in zeitlich versetzten Abständen irgendwelche Änderungen vornehmen. An diesem Punkt muss eine wichtige Entscheidung getroffen werden. Soll das Skript wie bisher weiterentwickelt und gepflegt werden? Oder sollen Methoden eingeführt werden, wie sie auch in der Software-Entwicklung üblich sind? Das bedeutet vor allem zwei Dinge: Eine Versionsverwaltung und das Testen einzelner Funktionalitäten mit Hilfe eines dafür vorgesehenen Werkzeugs. Das Thema Versionsverwaltung ist in Kap. 7 an der Reihe, das Thema Testen mit Hilfe des Pester-Moduls wird in diesem Kapitel vorgestellt. Pester ist ein Modul, das bei Windows Server 2016 und Windows 10 von Anfang an dabei ist. Es enthält eine Reihe von Functions, mit deren Hilfe sich Skripte, DSC-Ressourcen, Server-Konfigurationen und alles das, was per Befehlszeile abfragbar ist, automatisiert testen lassen. Das Besondere an den Functions ist, dass ihre Namen so gewählt wurden, dass sie an ein in der Software-Entwicklung übliches Testverfahren orientieren.

6.1 Was testet Pester?

Wer zum ersten Mal mit dem Begriff „testen“ konfrontiert wird, stellt sich im Allgemeinen viel zu viel darunter vor. Das Testen eines Skripts bedeutet in diesem Fall nicht, dass es unter Kontrolle des Testwerkzeugs ausgeführt wird und dieses auf geheimnisvolle Art und Weise entscheidet, ob das erwartete Verhalten auch tatsächlich eingetreten ist. Das Testen bezieht sich lediglich auf eine einzelne Befehlsfolge. Dabei wird getestet, ob das Ausführen der Befehlsfolge einen erwarteten und vorher festgelegten Rückgabewert liefert oder nicht. Ist es der Fall, gilt der Test als bestanden, ansonsten nicht.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_6

Im Zusammenhang mit einem PowerShell-Skript sind die zu testenden Befehlsfolgen in der Regel Functions, gegebenenfalls auch die Functions eines Moduls. Testen bedeutet, dass die Function mit bestimmten Argumenten ausgeführt und geprüft wird, ob ein erwarteter Zustand eintritt. Ein sehr einfaches Beispiel soll das Prinzip veranschaulichen. Eine Function, die zwei Zahlen addiert, muss, wenn sie mit den Werten 4 und 5 aufgerufen wird, den Wert 9 liefern. Tut sie das nicht, stimmt irgendetwas mit der Function (oder mit dem Test) nicht und der Autor erhält durch den nicht bestandenen Test einen Hinweis auf eine mögliche Fehlfunktion.

Ein etwas praxisnäheres Beispiel: Legt eine Function ein Benutzerkonto an und gibt im Erfolgsfall einen *$true*-Wert zurück, könnte ein Test darin bestehen abzufragen, ob der Rückgabewert *$true* oder *$false* ist. *Pester* fügt damit etwas zur PowerShell hinzu, das in der Welt der Software-Entwicklung seit vielen Jahren als Unit-Testing, zu Deutsch „Komponententests“, ein fester Bestandteil des Prozesses der Software-Entwicklung ist. Ein solcher Komponententest stellt sicher, dass eine einzelne Methode einen erwarteten Rückgabewert auch tatsächlich zurückgibt. Dank *Pester* lassen sich solche Tests auch mit PowerShell-Skripten und -Modulen und ihren Functions durchführen und in einen Bereitstellungsablauf integrieren.

6.1.1 Testen und DevOps

Das Schlagwort „DevOps“ beschreibt ein Umdenken bei der Umsetzung von Prozessen in der IT. Ein Aspekt ist, dass Ressourcen schneller und in kürzeren Zyklen bereitgestellt werden sollen. Geht es speziell um das Bereitstellen von Skripten, Modulen und DSC-Ressourcen, sorgt das Integrieren von Tests dazu, dass die Bereitstellung stärker automatisiert werden kann, indem die Tests ein Bestandteil des Bereitstellungsablaufs werden. Ein konkretes Beispiel: Eine DSC-Ressource wird nur dann bereitgestellt, wenn alle Tests bestanden werden. Damit ist gewährleistet, dass sie die gestellten Anforderungen erfüllt. Ein weiterer wichtiger Aspekt ist, dass sich nicht nur Befehle testen lassen, sondern beliebige Konfigurationszustände. Der Inhalt einer Konfigurationsdatei kann genauso getestet werden wie die Konfiguration eines Exchange-Servers oder der Inhalt eines Verzeichnisses. Das ist ein Grund, warum das Thema Testen in den letzten Jahren im IT-Bereich allgemein an Bedeutung zugenommen hat.

6.2 Das Pester-Modul im Überblick

Pester ist der Name eines Moduls, mit dem sich Functions in einem PowerShell-Skript automatisiert testen lassen. Auch wenn *Pester* nicht von Microsoft stammt, besitzt es einen halboffiziellen Stellenwert. *Pester* war bei Windows 10 zusammen mit der PowerShell 5.0 in der Version 3.4 von Anfang an dabei. Ihm gebührt die Ehre, das erste Open Source-Produkt zu sein, das jemals mit Windows ausgeliefert wurde. *Pester* wird von seinen Autoren laufend weiterentwickelt. Die aktuelle Version (Stand: Mai 2017) ist bereits eine

Version mit einer 4 in der Hauptversionsnummer. Das Modul umfasst 23 Functions. Die Adresse des Open Source-Projekts ist https://github.com/pester/Pester. Am einfachsten wird das Modul über das *Install-Module*-Cmdlet installiert. Unter Windows Server 2016 und Windows 10 sollte das bereits vorhandene Modul per *Update-Module*-Cmdlet aktualisiert werden.

- **Tipp** Der Umgang mit *Pester* ist in der Hilfe zu dem Modul ausführlich beschrieben, zum Beispiel „about_pester".

- **Tipp** Die *Pester*-Projektseite enthält eine kurze Einführung, die vor allem für den Einstieg im Umgang mit *Pester*, aber auch für den Praxisalltag bereits vollkommen ausreichend ist.

- **Tipp** Wer tiefer in das Thema einsteigen möchte, dem sei das eBook „The Pester book" von *Adam Bertram* und PowerShell Super-Guru *Don Jones* empfohlen. Wer sich für die Entstehungsgeschichte interessiert, findet einen kurzen Abriss in einem Blog-Eintrag des ursprünglichen Autors *Scott Muc* (http://scott-muc.com/growing-an-open-source-project-the-pester-story).

6.3 Die ersten Schritte mit Pester

Die ersten Schritte mit *Pester* sind einfach. Die größte „Schwierigkeit" besteht darin, sich generell an die Test-Philosophie und die etwas ungewöhnliche Syntax der *Pester*-Functions zu gewöhnen. Die wichtigste Formalität für den Einstieg ist, dass die Tests in einer separaten Skriptdatei abgelegt werden. Dabei gilt die Konvention, dass die Skriptdatei mit den Tests denselben Namen trägt wie die Skriptdatei, die getestet werden soll, und sich durch ein angehängtes „Tests" im Namen unterscheidet. Heißt eine Skriptdatei „Test1.ps1", heißt die Testskriptdatei „Test1.tests.ps1". Anhand dieser Konvention erkennt *Invoke-Pester* automatisch, welche Tests es ausführen soll. Diese Namenskonvention ist aber keine Voraussetzung. In der PowerShell ISE werden die Tests automatisch ausgeführt, wenn die Testdatei per [F5] gestartet wird.

Beispiel

Das folgende Beispiel geht von einem Skript „Test1.ps1" aus, das eine Function *Get-Zahl* enthält, die eine Zahl in einem Bereich zwischen 1 und einem Wert zurückliefert, der über einen optionalen Parameter festgelegt wird. Diese absichtlich sehr einfach gehaltene Function steht stellvertretend für eine x-beliebige Function, die eine x-beliebige Konfigurationsänderung durchführt und diese Änderung mit einem Rückgabewert quittiert.

```
function Get-Zahl
{
  param([Int]$Limit=10)
  Get-Random -SetSeed (Get-Date).Millisecond -Maximum $Limit -Minimum 1
}
```

Stellen Sie sich bitte für einen Augenblick vor, dass nicht bereits von Anfang an klar ist, dass die Function in jedem Fall einen Wert zwischen 1 und 10 zurückliefert und sich damit ein Test erübrigt.

Vorausgesetzt das *Pester*-Modul wurde hinzugefügt, lässt sich ein Test in Gestalt einer Befehlsfolge schreiben, welcher die *Get-Zahl*-Function ausführt und ihren Rückgabewert mit einem Wert vergleicht. Der Test befindet sich in der Datei „Test1.tests.ps1“. Der explizite Import des *Pester*-Moduls ist nicht erforderlich und soll lediglich abfragen, ob das Modul vorhanden ist.

```
Import-Module -Name Pester

describe "Allgemeine Tests" {

  .\Test1.ps1
  it "Should return a number less than 10" {
    (Get-Zahl) -lt 10 | Should be $true
  }
```

Der Aufbau der Befehlsfolge macht bereits deutlich, dass *Pester* ein wenig anders ist. Diese Andersartigkeit ist gewollt und basiert auf einer Philosophie, die als „Behavior Driven Development“, kurz BDD umbeschrieben wird. Ein Aspekt dieser Herangehensweise ist, dass die Testdefinition mit Hilfe natürlich sprachlich klingender Befehle umgesetzt wird. Durch diese Schreibweise wird das Verhalten der zu testenden Befehlsfolge und nicht ein Definitionsname in den Mittelpunkt der Testdefinition gestellt.

Describe, *it* und *should* sind lediglich die Namen von Functions aus dem *Pester*-Modul. Tab. 6.1 stellt die wichtigsten Functions des Moduls zusammen. Per *describe* wird der Name des Tests festgelegt und gleichzeitig ein Rahmen für die Ausführung der Testbefehle geschaffen. Innerhalb des Tests gibt es eine Reihe von *it*-Functions, auf die ebenfalls eine Beschreibung folgt, die den Sinn und Zweck des Tests auf den Punkt bringen. Die Function *should* prüft, ob ein beliebiger Ausdruck entweder *$true* oder *$false* ist. In dem konkreten Beispiel wird geprüft, ob *Get-Zahl* eine Zahl liefert, die kleiner 10 ist.

Um den Test auszuführen, gibt es zwei Möglichkeiten: In der ISE genügt das Ausführen des Testskripts per [F5], in der Konsole muss *Invoke-Pester* ausgeführt werden. Diese Function führt entweder alle Tests-Dateien im aktuellen Verzeichnis aus oder jene Tests, die über den *Name*- oder *Tag*-Parameter ausgewählt werden. War ein Test erfolgreich, wird der auf *describe* folgende Test in grün in der Konsole ausgegeben, ansonsten ist eine umfangreichere Fehlermeldung die Folge.

▶ **Hinweis** Pester-Tests können natürlich auch in Visual Studio Code ausgeführt werden. Allerdings muss dafür noch etwas Vorbereitungsaufwand in der Settings-Datei betrieben werden.

In der Praxis wird eine Befehlsfolge durch mehrere Tests getestet. Diese werden durch weitere *it*-Definitionen hinzugefügt.

Tab. 6.1 Die wichtigsten Functions des Pester-Moduls im Überblick

Function	Bedeutung
AfterAll	Legt innerhalb eines describe-Blocks eine Befehlsfolge fest, die nach allen Tests ausgeführt wird.
BeforeAll	Legt innerhalb eines describe-Blocks eine Befehlsfolge fest, die vor allen Tests ausgeführt wird.
AfterEach	Legt innerhalb eines describe-Blocks eine Befehlsfolge fest, die nach jedem einzelnen Test ausgeführt wird.
BeforeEach	Legt innerhalb eines describe-Blocks eine Befehlsfolge fest, die vor jedem einzelnen Test ausgeführt wird. Damit werden Befehle ausgeführt, die die Voraussetzungen für die Durchführung der Tests schaffen.
AssertMockCalled	Prüft, ob ein Mock eine bestimmte Anzahl oft ausgeführt wurde.
Assert-VerifiableMocks	Prüft, ob ein mit dem Verifiable-Parameter angelegter Mock eine bestimmte Anzahl oft ausgeführt wurde.
Context	Legt einen logischen Bereich innerhalb eines describe-Blocks an.
Describe	Legt einen logischen Bereich für einen oder mehrere it-Tests an. Der Bereich besitzt einen eigenen Scope für Mock-Commands und das TestDrive-Laufwerk. Der Bereich kann über die context-Function weiter unterteilt werden.
Invoke-Mock	Ruft einen Mock explizit auf.
Invoke-Pester	Ruft alle oder einzelne Tests einer Testdatei auf.
It	Leitet einen einzelnen Test ein.
Mock	Legt für ein Command eine Attrappe an.
New-Fixture	Legt ein Paar aus Ps1-Datei und Tests.ps1-Datei mit einem Testgerüst an.
Should	Prüft eine einzelne Voraussetzung (Assertion).

Beispiel

Im folgenden Beispiel prüft ein weiterer Test, ob die Function *Get-Zahl* eine Zahl liefert, die kleiner 20 ist, wenn die Function mit dem *Limit*-Parameter aufgerufen wird.

```
it "Should return a number less than 20" {
    (Get-Zahl -Limit 20) -lt 20 | Should be $true
}
```

Wird der Test ausgeführt, wird auch der zweite Test bestanden. Stellen wir uns für einen Moment vor, wir wären nicht sicher, wie der Parameter *Limit* bei *Get-Zahl* Zahl funktioniert. Bedeutet eine Obergrenze von 20, dass Zahlen bis oder einschließlich 20 geliefert werden? Natürlich kann man in der Hilfe nachsehen, doch nur selten ist das Verhalten einer Befehlsfolge zu 100 % klar definiert. Auch ein Test kann diese Frage nicht direkt beantworten. Es ist aber mit wenig Aufwand möglich, die Function eine große Anzahl oft aufzurufen. Dabei wird geprüft, ob die Obergrenze bei einem dieser Aufrufe zurückgegeben wird. Ist das nicht der Fall, kann man mit hoher Wahrscheinlichkeit davon ausgehen, dass die Obergrenze selber nicht als Zahl geliefert wird.

Beispiel

Der folgende Test prüft, ob die Function *Get-Zahl* die übergebene Obergrenze auch als Wert zurückgibt. Dies wird dadurch getestet, indem die Function eine große Anzahl wiederholt ausführt und bei jedem Aufruf der Rückgabewert mit der Obergrenze verglichen wird. Das Ergebnis, das entweder *$true* oder *$false* ist, wird auf eine Variable aufaddiert. Ist diese am Ende 0, gilt der Test als bestanden.

```
describe "Reihen-Tests" {
  it "Should return a number less than 10" {
    $Ergebnis = $false
    1..1000 | ForEach {
        $Ergebnis += (Get-Zahl) -ge 10
    }
    $Ergebnis -eq 0 | Should be $true
  }
}
```

Das Beispiel soll deutlich machen, dass es theoretisch unendlich viele Möglichkeiten gibt, einen Test zu schreiben. Außerdem: Auch Tests müssen getestet werden, bis sie eine Befehlsfolge so testen, dass ein valides und reproduzierbares Resultat entsteht.

6.4 Einen Test-Rahmen mit New-Fixture anlegen

Die Function *New-Fixture* legt einen Rahmen für einen Test an. Dieser besteht aus einer Ps1-Datei und einer Datei, die denselben Namen trägt wie die Ps1-Datei, nur mit einem „Tests“ am Ende des Dateinamens. Auch wenn diese Function in vielen Einführungen zu *Pester* verwendet wird, ist sie lediglich eine Option. Sie soll den Einstieg in *Pester* etwas erleichtern.

Beispiel

Der folgende Aufruf von *New-Fixture* hat zur Folge, dass im aktuellen Verzeichnis die Dateien „Test.ps1“ und „Test.Tests.ps1“ angelegt werden. Die Tests-Datei enthält Befehle, durch die automatisch jene Ps1-Datei ausgeführt wird, die denselben Namen trägt wie die Tests-Datei, aber ohne das „Tests“ im Namen (der Name der Variablen *sut* steht für „Script Under Test“, ein im Bereich des Software-Testens geläufiger Begriff).

```
New-Fixture –Name Test
```

Damit kann mit *Invoke-Pester* bereits ein erster Probelauf durchgeführt werden. Der Aufruf erfolgt im Allgemeinen ohne Parameter.

```
Invoke-Pester
```

Die umfangreiche Fehlermeldung hat eine einfache Erklärung. Der Aufruf war erfolgreich, doch da die getestete Function nicht den erwarteten Wert geliefert hat, ist (natürlich) ein Fehler die Folge.

Die letzten beiden Zeilen der Ausgabe sind der „Beweis“ dafür, dass der Test erfolgreich absolviert wurde:

```
Tests completed in 649ms
Passed: 0 Failed: 1 Skipped: 0 Pending: 0 Inconclusive: 0
```

Möchten Sie erreichen, dass der Test erfolgreich ist, müssen Sie lediglich den Testaufruf in der Datei „Test.tests.ps1“ anpassen und dafür sorgen, dass der per „Should Be“ getestete Wert jenem Wert entspricht, der über die Pipeline übergeben wird.

Beispiel

Das folgende Beispiel passt den Test an, so dass er erfolgreich absolviert wird.

```
Describe "test" {
  It "does something useful" {
    $true | Should Be $true
  }
}
```

Jetzt hat der erneute Aufruf von *Invoke-Pester* einen positiven Bescheid zur Folge:

```
Tests completed in 93ms
Passed: 1 Failed: 0 Skipped: 0 Pending: 0 Inconclusive: 0
```

6.5 Test-Ergebnisse vergleichen mit Should – die Rolle der Assertions

Pester verfolgt eine eigene Philosophie bei der Art und Weise, wie eine Test-Function aufgebaut sein muss. Im Mittelpunkt stehen umgangssprachliche Namen für die Elemente eines Tests wie *describe*, *it* und *should*.

Beispiel

Das folgende Beispiel soll die Bedeutung von *it* veranschaulichen. In dem auf ein *it* die umgangssprachliche Beschreibung des Tests folgt, wird die Bedeutung dieses Tests besser deutlich. Dass es sich bei *it* um eine Function handelt, tritt dabei in den Hintergrund. Der Umstand, dass die Parameter *Name* und *Test* weggelassen werden, verstärkt die Wirkung auf den Betrachter.

```
it "Passes when user account does not exist and 'Ensure' is 'Absent'" {
  Test-TargetResource @testAbsentParams | Should Be $true;
}
```

Die Verwendung umfangssprachlicher Begriffe hat aber auch Nachteile. Für einen erfahrenen PowerShell-Anwender wird es etwas schwerer, die erlaubte Syntax für die Elemente eines Tests herauszufinden. Das wird am Beispiel von *should* deutlich. Offiziell ist es wie *it* eine Function, doch ein „Get-Command should -syntax“ liefert

kein Ergebnis. Die Function besitzt keine Parameter. Erst die Hilfe zu *should*, die über ein „help about_should“ aufgerufen wird, beschreibt die Rolle von *should* ausführlich. Offiziell ist es eine Function, die den Inhalt der Pipeline über sogenannte Assertions prüft, die auf die Function folgen. Eine „Assertion“ (zu Deutsch „Zusicherung“) ist eine Aussage über einen Zustand, wie zum Beispiel den Wert einer Variablen oder den Rückgabewert einer Function. *Should* prüft mit Hilfe solcher Assertions, ob die Aussage erfüllt ist oder nicht. Ist eine Assertion erfüllt, gilt der Test als bestanden. Ansonsten ist ein Fehler vom Typ *PesterFailure* die Folge. Ein Beispiel für eine Assertion ist *be*. Tab. 6.2 stellt die Assertions zusammen, die auf ein *should* folgen dürfen.

Beispiel

Der folgende Test prüft, ob der Aufruf der Function *Get-CriticalProcess* weniger als 10 Prozesse liefert. Er wird aber noch nicht zum gewünschten Ergebnis führen.

```
Get-CriticalProcess | Should BeLessThan 10
```

Doch diese Syntax funktioniert nicht, auch wenn der Test logisch erscheint. Der Grund ist natürlich, dass die Function eine bestimmte Anzahl an Prozessobjekten liefert, *should* diese aber nicht automatisch zählt, auch wenn als Assertion *BeLessThan* gewählt wurde. Das Zählen muss bereits durchgeführt worden sein.

Tab. 6.2 Should und die in Frage kommenden Assertions

Assertion	Bedeutung
Be	Vergleicht den Inhalt der Pipeline mit einem Wert und löst eine Ausnahme aus, wenn beide Werte nicht gleich sind.
BeExactly	Prüft auf Gleichheit unter Berücksichtigung der Groß-/Kleinschreibung.
BeGreaterThan	Führt einen Zahlenvergleich vom Typ größer durch.
BeLessThan	Führt einen Zahlenvergleich vom Typ kleiner durch.
BeLikeExactly	Führt einen Vergleich mit dem Platzhalter * unter Berücksichtigung der Groß-/Kleinschreibung durch.
BeOfType	Prüft, ob das Objekt einen bestimmten Typ besitzt.
BeNullOrEmpty	Prüft, ob das Objekt in der Pipeline einen Wert besitzt.
Exist	Prüft, ob das Objekt in der Pipeline auf einem PSDrive existiert. Dieser Vergleich wird daher zum Beispiel mit Verzeichnissen und Dateien durchgeführt.
Contain	Prüft, ob eine Datei einen bestimmten Inhalt besitzt.
Match	Prüft den Inhalt der Pipeline mit Hilfe eines regulären Ausdrucks.
Throw	Prüft, ob der zuvor ausgeführte Befehl eine Exception geworfen hat. In diesem Fall müssen der Befehl oder die Befehlsfolge vor dem Pipe-Operator in geschweifte Klammern gesetzt werden. Mit *Not Throw* wird auf keine Exception geprüft.
Not	Kehrt die Bedeutung einer Assertion um.

Beispiel

Der folgende Test entspricht dem letzten Beispiel, nur dass dieses Mal die Anzahl der kritischen Prozesse vor der Assertion abgefragt wird.

```
(Get-CriticalProcess).Count | Should BeLessThan 10
```

Dieser Test ist immer dann erfüllt, wenn die Anzahl der Prozesse, die von Get-CriticalProcess geliefert werden, kleiner als 10 ist.

6.6 PowerShell-Commands nachbilden über Mocks

Nicht immer können oder sollen die getesteten Commands wirklich ausgeführt werden. Entweder kann das Command nicht ausgeführt werden, zum Beispiel weil eine Datenbank nicht zur Verfügung steht oder weil der Test auf einem Computer ohne Netzwerkanbindung ausgeführt wird. Oder das Command soll nicht ausgeführt werden, da es zum Beispiel etwas entfernt, das nach jedem Testlauf wiederhergestellt werden müsste. Für diese Situationen gibt es die „Mocks“, zu Deutsch „Attrappen“. Ein Mock beziehungsweise ein Mock-Command bildet ein existierendes Command bezüglich seiner Parameter nach, so dass das Mock-Command anstelle des Original-Commands ausgeführt wird. Es lässt sich festlegen, was bei der Ausführung des Mock-Commands passieren soll. Außerdem kann festgelegt werden, dass das Mock-Command nur dann anstelle des Original-Commands ausgeführt wird, wenn bestimmte Parameter und/oder Parameterwerte übergeben werden. Und es lässt sich abfragen, wie oft ein Mock-Command bei der Ausführung des Tests ausgeführt wurde.

Das „Mocken“ von Commands geht bei *Pester* dank der *Mock*-Function sehr einfach. Im einfachsten Fall wird sie lediglich mit dem Namen des Commands aufgerufen, für das eine Attrappe benötigt wird.

▶ **Hinweis** Voraussetzung für ein Mocken ist, dass das Command in der PowerShell-Sitzung vorhanden ist. Möchte man zum Beispiel die AD-Cmdlets auf einem Notebook mit Windows 10 testen, müssen zuvor die *Remote Server Administration Tools* (RSAT) installiert werden, damit die AD-Cmdlets vorhanden sind. Ansonsten müsste für jedes AD-Cmdlet eine leere Function definiert werden, die alle Parameter enthält, die bei dem zu testenden Aufruf verwendet werden. Die *Mock*-Function benötigt das zu „mockende“ Kommando als Vorbild, um darüber das Mock-Command nachbauen zu können.

Beispiel

Im folgenden Beispiel soll eine etwas umfangreichere Function getestet werden, die im Active Directory per *New-AdUser* ein Benutzerkonto anlegt und dieses per *Add-AdGroupMember* zu einer Sicherheitsgruppe hinzufügt. Gibt es die Gruppe noch nicht, wird sie per *New-AdGroup* angelegt. Die Abfrage, ob die Gruppe existiert, wird per *Get-AdGroup* durchgeführt. Am Ende wird die Befehlsfolge entweder ausgeführt oder es wird eine Exception ausgelöst. Ein erfolgreicher Test bedeutet damit, dass alle drei beziehungsweise vier AD-Cmdlets ausgeführt werden konnten.

```
function Create-UserAccount
{
  param([String]$Accountname, [String]$Groupname="PsKurs")

  New-ADUser -Name $Accountname
  if ((Get-ADGroup -Identity $Groupname -ErrorAction Ignore).Name -ne
$Groupname)
  {
    New-ADGroup -Name $Groupname -GroupCategory Security -GroupScope
DomainLocal
  }
  Add-ADGroupMember -Identity $Groupname -Members $Accountname
}
```

In der Praxis wird diese einfache Befehlsfolge immer dann funktionieren, wenn der Zugriff auf das Active Directory mit den Berechtigungen eines Domänadmins möglich ist und das Benutzerkonto noch nicht vorhanden ist. Ist die Function Teil eines größeren Skripts, ist es wünschenswert, dass sie automatisiert getestet werden kann.

Beispiel

Der folgende Test testet die Function *Create-UserAccount*, indem ein Testkonto mit dem Namen „TestKonto“ angelegt wird.

```
Describe "Benutzekonten-Tests" {

  it "Tests creating a new user account and adding it to a group" {
    { Create-UserAccount -AccountName TestKonto } | Should Not Throw
  }
}
```

Wird der Test von einem Benutzer ausgeführt, der nicht an der Domäne angemeldet ist, wird er zwangsläufig scheitern, da bereits *New-ADUser* eine Exception auslöst. Die Lösung ist, alle AD-Cmdlets durch Mocks zu ersetzen.

Beispiel

Das folgende Beispiel erweitert den Test aus dem letzten Beispiel, indem für jedes der insgesamt vier AD-Cmdlets *New-ADUser*, *Get-AdGroupMember*, *New-AdGroup* und *Add-AdGroupMember* jeweils ein Mock angelegt wird.

```
Describe "AD-Tests mit Mock" {
  . .\ADUserAnlegen.ps1

 Context "Mock it" {
  Mock -CommandName New-ADUser
  Mock -CommandName Add-ADGroupMember
  Mock -CommandName Get-AdGroup
  Mock -CommandName New-ADGroup

  it "Tests creating a new useraccount with Mocks" {
    { Create-UserAccount -AccountName TestKonto } | Should not throw
  }
 }
}
```

Kommt es darauf an, dass das „gemockte“ Command etwas zurückgibt, kommt der *MockWith*-Parameter ins Spiel. Über ihn wird ein Scriptblock festgelegt, der bei der Ausführung des Mocks ausgeführt wird. Hier kann ein beliebiger Rückgabewert erzeugt werden.

6.6.1 Feststellen, ob ein Mock-Command ausgeführt wurde

Möchte man feststellen, ob und wie oft ein Mock-Command ausgeführt wird, geschieht das mit der Function *Assert-MockCalled*. Über den *Times*-Parameter kann eine Zahl angegeben werden. Sie legt fest, wie oft das Mock-Command aufgerufen worden sein muss. Auf diese Weise lässt sich nachvollziehen, ob die Mock-Commands tatsächlich zum Einsatz kamen.

6.7 Tests in einen Ablauf einbeziehen

Soll ein Test Teil eines Ablaufs sein, gibt es mindestens zwei Optionen. Die erste Option besteht darin, beim Aufruf von *Invoke-Pester* den *PassThru*-Parameter zu setzen. Er bewirkt, dass anstelle einer Konsolenausgabe ein Objekt zurückgegeben wird, über das sich unter anderem die Anzahl der erfolgreichen und fehlgeschlagenen Tests abfragen lässt. Die zweite Option ist der *Enable-Exit*-Parameter. Er bewirkt, dass der PowerShell-Prozess, in dem *Invoke-Pester* ausgeführt wird, beendet wird und die Anzahl der fehlgeschlagenen Tests als Returncode zurückgegeben wird. Der Rückgabewert kann in der PowerShell-Konsole über die Variable *LastExitCode* und in einer Stapeldatei über *ERRORLEVEL* abgefragt werden.

Auf diese Weise lässt sich der Aufruf auch in einem Build-Ablauf (zum Beispiel MsBuild) integrieren. Im einfachsten Fall wird eine Stapeldatei ausgeführt, die über *Powershell.exe* das Testskript ausführt. Über den Returncode von *Powershell.exe* beziehungsweise der Stapeldatei wird entschieden, ob die Tests erfolgreich waren oder nicht.

6.8 Testen als (Lebens-)Philosophie

Die kleinen Beispiele in diesem Abschnitt haben deutlich gemacht, dass der Umgang mit dem *Pester*-Modul grundsätzlich einfach ist. Nicht oder nur ansatzweise beantwortet haben sie Fragen, warum Tests wichtig sind und welche Vorgehensweise man für das Schreiben von Tests wählen sollte. Viele PowerShell-Anwender dürften zunächst keinen Sinn darin sehen, Testskripte zu schreiben, die lediglich testen, ob zum Beispiel eine Function einen Wert zurückgibt, der sich bereits durch Nachvollziehen der Befehlsfolge der Function ergibt. Es bleiben daher am Ende des Kapitels noch einige grundsätzliche Fragen offen.

Auf alle Fragen gibt es Antworten, die sich nach dem richten, was getestet werden soll und welche Rolle ein Test in einem Arbeitsablauf spielt. Bestehende Skripte sind in der Regel nicht automatisch testfähig. Damit ein Skript „testfähig“ wird, müssen seine Funk-

tionalitäten möglichst granular auf Functions verteilt werden, die einzeln getestet werden. Zwischen den Functions sollten keine Abhängigkeiten bestehen, so dass der Test von Function A nicht voraussetzt, dass zuvor eine Function B aufgerufen wurde.

Der Hauptgrund für Tests ist, dass Tests dazu beitragen, Fehler möglichst früh im Entwicklungszyklus zu entdecken. Dieser Aspekt erhält bei der Entwicklung von PowerShell-Skripten immer dann eine Bedeutung, wenn die Qualitätssicherung ein Thema ist. Soll ein Skript nur eine bestimmte, eng abgegrenzte Aufgabe erfüllen und es wird von einem kleinen Personenkreis oder nur einer Person eingesetzt, spielt das Thema Qualitätssicherung und damit auch das von Tests keine Rolle.

Sobald ein Skript von einem größeren Personenkreis eingesetzt wird und kritische Aufgaben übernimmt, kommt es darauf an, dass es fehlerfrei ausführt, es zum gewünschten Resultat führt und keine Nebeneffekte besitzt, die sich eventuell erst nach längerem Einsatz bemerkbar machen. Tests tragen dazu bei, dass Fehler reduziert und Nebeneffekte ausgeschlossen werden. Alleine durch den Umstand, dass man sich beim Schreiben von Test-Functions intensiv mit den zu testenden Functions beschäftigt und einzelne Functions so umstellt, dass sie sich sinnvoll testen lassen, entdeckt man bereits Fehler in den zu testenden Funktionalitäten.

6.9 Zusammenfassung

Sollen die Functions eines Skripts oder Moduls bezüglich ihres Rückgabewertes automatisiert getestet werden, wird dafür das *Pester*-Modul verwendet, das ab der Version 5.0 der PowerShell und damit bei Windows 10 von Anfang an dabei ist. Das Modul umfasst eine Reihe von Functions, deren Namen an die Art und Weise angelehnt wurden wie im Rahmen des „Behavorial Driven Test“-Prinzips, kurz BDD, Tests durchgeführt werden. Die wichtigsten Functions sind *describe* für das Anlegen eines Testrahmens, *it* für das Anlegen eines einzelnen Tests und *should* für den Vergleich des Rückgabewertes mit einem erwarteten Wert. Kann oder soll ein Command während des Tests nicht ausgeführt werden, wird es mit der *Mock*-Function durch eine Attrappe ersetzt, die anstelle des Commands ausgeführt wird. Pester wird laufend weiterentwickelt, so dass zum Zeitpunkt, an dem Sie diese Zeilen lesen, bereits eine aktuellere Version in der PowerShell Gallery zur Verfügung stehen dürfte.

Skripte und Module bereitstellen

7

Zusammenfassung

In diesem Kapitel stelle ich Möglichkeiten vor, die erfahrenen PowerShell-Anwendern zur Verfügung stehen, wenn es um eine zentrale Ablage für Skripte und Module geht. Mit den Modulen *PackageManagement* und *PowerShellGet*, die seit der Version 5.0 ein fester Bestandteil der PowerShell sind, bietet sich eine naheliegende Vorgehensweise bereits an. Mit Hilfe der Cmdlets und Functions aus den beiden Modulen kann ein PowerShell-Anwender sowohl über die Konsole als natürlich auch im Rahmen eines Skripts Module, Skripte und DSC-Ressourcen von einer zentralen Ablage abrufen und lokal abspeichern. Eine Suchfunktion erlaubt die Suche unter Einbeziehung von Metadaten wie einem Stichwort oder dem Autor. Die Ablage ist zum Beispiel die PowerShell Gallery oder ein eigenes Repository im Intranet. Die Paketverwaltung der PowerShell ist natürlich nur eine Option. Im einfachsten Fall werden Modul- und Skriptdateien per *Invoke-WebRequest*-Cmdlet (Wget) aus einem Webverzeichnis geladen oder per Ftp übertragen.

Alle Hinweise für Module gelten (natürlich) auch für DSC-Ressourcen, die ebenfalls als Module vorliegen.

7.1 Die PowerShell-Paketverwaltung im Überblick

Mit Windows 10 gibt es bei Windows erstmals eine Paketverwaltung, die über die Befehlszeile angesprochen wird. Bei Windows Server 2016 gibt es sie auch. So etwas kannte man bislang nur von Betriebssystemen wie *Linux* und *macOS* (vormals OS X). Die neue Paketverwaltung soll das Installieren beliebiger Anwendungen über die Befehlszeile ermöglichen (vergleichbar mit *apt* und *apt-get* oder *yum* unter Linux). Soll zum Beispiel

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_7

der Zip-Manager 7zip installiert werden, wird dies per „Install-Package 7Zip“ erledigt. Die Paketverwaltung von Windows basiert auf dem *PackageManagement*-Modul des WMF 5.0, das bei Windows 10 von Anfang an dabei ist. Dahinter steckt sowohl eine Intrastruktur für die Verwaltung im Grunde beliebiger Paketmanager als auch ein Satz von Cmdlets, über den ein Paketmanager angesprochen wird.

Eine Paketverwaltung verwaltet Pakete, so viel ist bereits klar. Doch was muss man sich unter einem Paket vorstellen? Der Begriff „Paket“ oder „Package“ besitzt in der IT-Welt eine lange Tradition. Er beschreibt alle Dateien, die für die Installation einer Anwendung erforderlich sind. Im einfachsten Fall besteht ein Paket aus einer einzelnen Msi-Datei, sowie einer Datei, die den Inhalt des Pakets beschreibt und zum Beispiel die Art der Installation festgelegt. Das Thema Paketverteilung ist eines der Schwerpunktthemen im modernen IT-Alltag. Das *PackageManagement*-Modul kann und soll zwar nicht mit den Schwergewichten der Branche konkurrieren, zumal es ausschließlich auf einem Pull-Modell basiert, für das Verteilen kleiner Anwendungen ist es aber gut geeignet.

7.1.1 Ein Blick hinter die Kulissen

Die mit WMF 5.0 eingeführte Paketverwaltung ist eine Infrastruktur, die theoretisch beliebige Sorten von Paketverwaltungen unter einem Dach zusammenfasst und diese über einen einheitlichen Satz an Cmdlets aus dem *PackageManagement*-Modul ansprechbar macht. *PackageManagement* ist inzwischen ein eigenständiges Open Source-Projekt, das auf einem eigenen Projektportal betrieben wird: https://github.com/oneget. Der Name „OneGet“ war der ursprüngliche Projektname, der auch in den ersten Vorabversionen von WMF 5.0 als Modulname verwendet wurde, aus internen Gründen aber nicht beibehalten werden konnte. Der neue Name des Projekts ist einfach „PackageManagement“. Die PowerShell verfügt über zwei Module: *PackageManagement* und *PowerShellGet*. Letzteres verwendet *PackageManagement*, um Module, DSC-Ressourcen und Skripte aus einem Repository abrufen oder in ein solches veröffentlichen zu können.

Den Entwicklern des *PackageManagement*-Projekts war klar, dass es für Windows in Zukunft nicht eine, sondern mehrere Paketverwaltungen geben würde. Außerdem mussten sie den Umstand berücksichtigen, dass Windows-Anwender seit 1985 ohne eine Paketverwaltung ausgekommen sind, und dass daher zum Beispiel auch per Msi-Installer installierte Anwendungen ansprechbar sein sollten. Und noch eine Überlegung floss in die Entwicklung ein: Es soll bei einer universellen Paketverwaltung nicht nur um klassische Anwendungen gehen, sondern um alles, was sich im weiteren Sinne als „Paket“ verteilen lässt. Dazu gehören unter anderem Images für einen Nanoserver, Docker-Container, Github Gist-Dateien und natürlich PowerShell-Module und -Skripte.

Die *PackageManagement*-Entwickler bezeichnen ihre Software daher auch als „Package Management Aggregator“.[1] Im Mittelpunkt stehen Provider, die eine oder

[1] Eine offizielle Abkürzung, wie etwa PMA, hat man sich dafür aber nicht einfallen lassen.

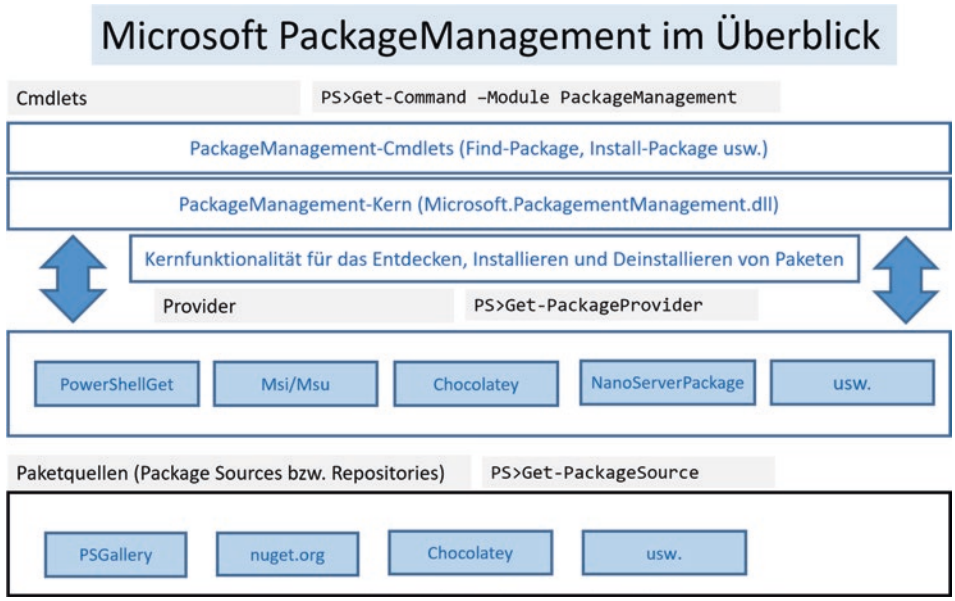

Abb. 7.1 Die Package Management-Infrastruktur (OneGet) von Microsoft im Überblick

mehrere Paketquellen („Package Source“) ansprechen und das Hinzufügen von Paketen über diese Quelle ermöglichen. Die Paketquellen heißen beim *PowerShellGet*-Modul „Repositories“. Beispiele für Package-Provider sind *Msi* (Anwendungen), *Msu* (Updates), *Programs* (die Anwendungen aus den Programme-Verzeichnissen) und *NuGet* (für Pakete, die direkt über *nuget.org* geladen werden). Abb. 7.1 stellt das PackageManagement-Konzept in einem Schaubild zusammen.

Die Rolle von NuGet

Beim ersten Aufruf von *Install-Package* oder *Install-Module* werden Sie aufgefordert, den *NuGet*-Package-Provider zu installieren. Dieser wird in Gestalt der Assembly *Microsoft.PackageManagement.NuGetProvider.dll* entweder im Verzeichnis *C:\Program Files\PackageManagement\ProviderAssemblies* oder unter *$env:localappdata\Packagemanagement\ProviderAssemblies* abgelegt. *NuGet* ist der Name des bereits vor vielen Jahren eingeführten Paketmanagers von Microsoft. Es ist eine Open Source-Anwendung, die auf der .NET-Laufzeit basiert. Anfangs wurde *NuGet* im Rahmen von Visual Studio eingesetzt, bei der PowerShell spielt der Paketmanager inzwischen ebenfalls eine wichtige Rolle, da über ihn Anwendungspakete und Module geladen werden (Abb. 7.2). Auf der Webseite http://nuget.org steht das Konsolenprogramm *Nuget.exe*, über das sich Packages direkt über die Befehlszeile laden lassen, zum Download zur Verfügung. Diese Pakete stammen aber ausschließlich vom *NuGet*-Portal und sind in erster Linie Erweiterungen, die Entwickler für ihre Anwendungen benötigen.

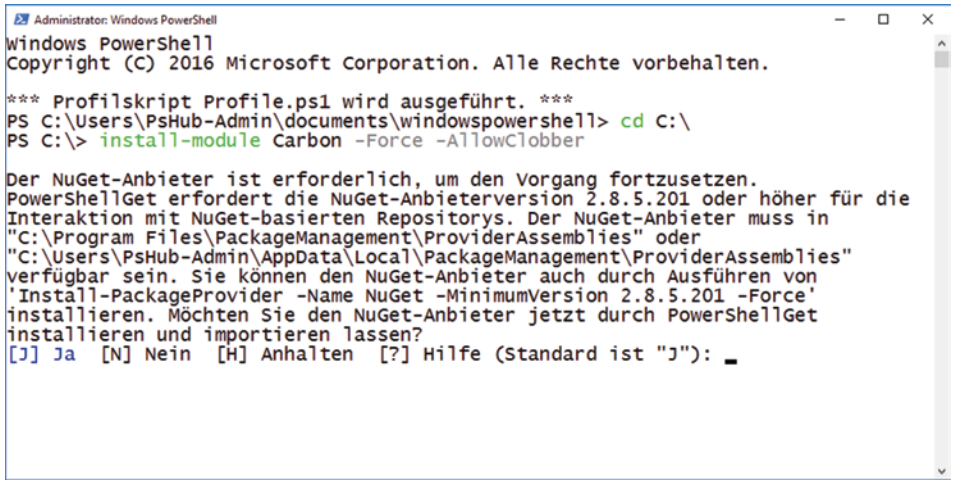

Abb. 7.2 Die Modulverwaltung setzt auf dem NuGet-Paketmanager auf

▶ **Hinweis** Innerhalb von Visual Studio wird im Fenster des Paketmanagers ein eigener PowerShell-Host geladen, in dem eine Reihe von NuGet-Cmdlets wie zum Beispiel *Install-Package* angeboten werden. Diese haben nichts mit den Cmdlets aus dem *PackageManagement*-Modul zu tun.

▶ **Hinweis** Der ganz große Wurf ist den Entwicklern des *PackageManagement*-Moduls aus der Sicht des Autors leider (noch) nicht gelungen. Die Kritikpunkte sind allerdings eher Kleinigkeiten wie eine nicht konsistente Terminologie (Paketquelle und Repository), eine etwas verwirrende Auswahl an Parametern bei einigen Cmdlets, eine funktionale Überschneidung zwischen den Modulen *PackageManagement* und *PowerShellGet* und der Umstand, dass das Zusammenspiel zwischen der PackageManagement-Infrastruktur und externen Providern, die meistens von einzelnen Anwendern stammen, zwangsläufig fehlerträchtig ist und die resultierenden Fehlermeldungen schlecht bis gar nicht dokumentiert sind, so dass sich die Fehlerursache nicht so einfach nachvollziehen lässt. Hinzu kommt, dass der Bedarf für eine Paketverwaltung in der Windows-Welt offenbar nicht allzu groß ist, da im Internet relativ wenig Anwenderfeedback existiert.[2] Das *PackageManagement*-Projekt wird laufend weiterentwickelt, so dass es in Zukunft konsistenter und funktional reichhaltiger werden dürfte. Da es ein Open Source-Projekt ist, kann jeder PowerShell-Anwender dazu beitragen und sei es durch das Melden von Fehlern im Projektportal und durch Verbesserungsvorschläge.

[2] Was sich natürlich noch ändern kann. Die PowerShell hat auch ein paar Jahre benötigt, bis sie tatsächlich von einem größeren Kreis von Administratoren tatsächlich eingesetzt wurde.

Mit WMF 5.1 wurden beim *PackageManagement*-Modul kleinere Verbesserungen eingeführt. Die wichtigste ist, dass das Hinzufügen von Paketen dank dem *Source*-Parameter beim *Install-Package*-Cmdlet auch auf Computern ohne Internet-Anbindung funktioniert, indem ein bereits heruntergeladenes Paket verwendet wird. Zwei weitere Verbesserungen betreffen den Umstand, dass Aktivitäten des Paketmanagers im PowerShell-Ereignisprotokoll (*Microsoft-Windows-PowerShell/Operational*, die Einträge besitzen die ID 4101) protokolliert werden, und dass ein Proxy-Server über die Parameter *Proxy* und *ProxyCredential* angesprochen werden kann.

7.1.2 Überblick über das PackageManagement-Modul

Das im Zusammenhang mit der Version 5.0 eingeführte *PackageManagement*-Modul umfasst insgesamt 13 Cmdlets für eine Paketverwaltung. Tab. 7.1 stellt die Cmdlets des Moduls zusammen.

7.1.3 Die ersten Schritte mit der Paketverwaltung

Voraussetzung für das Installieren eines Pakets ist, dass mindestens ein Provider vorhanden ist. Ein *Get-PackageProvider* listet alle vorhandenen Provider auf. Über *Install-PackageProvider* wird ein Provider hinzugefügt. Da die Provider über die PowerShell Gallery zur Verfügung stehen und der erforderliche Provider von Anfang an dabei ist, ist das Hinzufügen weiterer Provider grundsätzlich kein Problem. Tab. 7.2 stellt die für die PowerShell-Praxis interessanten Package-Provider zusammen. Die Liste erhebt keinen Anspruch auf Vollständigkeit. Da sich eigene Provider mit relativ wenig Aufwand entwickeln lassen, ist die Wahrscheinlichkeit groß, dass sich die Liste zum aktuellen Zeitpunkt bereits erweitert hat.

Das *Get-PackageSourc*e-Cmdlet listet die vorhandenen Paketquellen auf. Anders als man es vermuten könnte, ist die Auswahl der Paketquellen am Anfang überschaubar: Es gibt lediglich *PSGallery*, die *PowershellGet* als Provider verwendet. Möchte man zum Beispiel eine Anwendung über ein *Install-Package* hinzufügen, muss dazu erst eine passende Paketquelle hinzugefügt werden.

7.1.4 Anwendungspakete über Chocolatey installieren

Wer per PowerShell und dem *Install-Package*-Cmdlet Anwendungen installieren möchte, benötigt dazu *Chocolatey*. *Chocolatey* ist der Name einer eigenständigen Paketverwaltung für Windows und eines dazugehörigen Portals *chocolatey.org*, auf dem mehrere tausend Packages für bekannte und weniger bekannte Windows-Anwendungen zur Verfügung stehen.[3]

[3] Stand März 2017 waren es 4622 Pakete, wobei natürlich Quantität nicht gleich Qualität bedeutet, da jedes Paket gepflegt werden muss, damit stets die aktuelle Version installiert wird und Fehler behoben werden.

Tab. 7.1 Die Cmdlets aus dem PackageManagementModul im Überblick

Cmdlet	Was macht es?
Find-Package	Lokalisiert das gesuchte Paket entweder in allen vorhandenen oder nur in der bzw. den angegebenen Paketquelle(n).
Find-PackageProvider	Gibt die zurzeit in der PSGallery verfügbaren Paketprovider aus.
Get-Package	Gibt die installierten Pakete aus, die über alle oder die angegebenen Paketprovider installiert wurden. Ein „Get-Package -ProviderName Programs“ gibt die installierten Anwendungen aus.
Get-PackageProvider	Gibt die aktuell geladenen Paketprovider aus. Der Parameter ListAvailable gibt auch die vorhandenen Provider aus.
Get-PackageSource	Gibt die vorhandenen Paketquellen aus.
Import-PackageProvider	Fügt einen vorhandenen Paketprovider zur aktuellen Sitzung hinzu. Der Name des Providers muss nicht mit dem Modulnamen identisch sein, in dem er sich befindet.
Install-Package	Fügt ein Paket lokal hinzu, das von den bereits eingerichteten Paketquellen zur Verfügung gestellt wird. Mit WMF 5.1 wurde der *Source*-Parameter eingeführt, so dass auch ein Paket von einem Netzwerklaufwerk geladen werden kann.
Install-PackageProvider	Fügt einen Paketprovider hinzu.
Register-PackageSource	Registriert eine neue Paketquelle.
Save-Package	Speichert ein Paket auf dem lokalen Computer in dem angegebenen Verzeichnis, ohne es zu installieren. Muss von einem Provider unterstützt werden, *Chocolatey* bietet diese Option aktuell (Stand: März 2017) nicht.
Set-PackageSource	Ändert Einstellungen bei einer Paketquelle, zum Beispiel ihre Vertrauenswürdigkeit.
Uninstall-Package	Entfernt ein Paket, indem die hinzugefügten Dateien gelöscht werden.
Unregister-PackageSource	Entfernt die angegebene Paketquelle. Das Pendant zu Register-PackageSource. Der Name der Quelle wird über den *Source*-Parameter angegeben. Die Paketquelle wird ohne Bestätigungsanforderung entfernt.

Tab. 7.2 Package Provider, die entweder von Anfang an zur Verfügung stehen oder nachträglich installiert werden müssen

Provider	Was wird geholt?
Msi	Lokale Programme und allgemein Dateien, die per Msi-Installer installiert wurden.
Msu	Lokal installierte Updates.
NuGet	Packages, die über einen NuGet-Server zur Verfügung gestellt werden.
Programs	In erster Linie lokale Anwendungen.
PowerShellGet	Module, Skripte und DSC-Ressourcen, die von einem NuGet-Server zur Verfügung gestellt werden.
WSAProvider	Windows Server Apps (Anwendungen für Nanoserver).

Chocolatey funktioniert auch unabhängig von der PowerShell über ein Befehlszeilentool mit dem Namen *Choco.exe* und einem Satz von Kommandos, die, ähnlich wie bei *Nuget.exe*, für die Paketverwaltung zur Auswahl stehen. Dieses muss über die Webseite https://chocolatey.org zuerst installiert werden. Ein „choco install vivaldi" lädt die beliebte Browser-Alternative als Package herunter und installiert die Anwendung über ein PowerShell-Skript (in der Regel wird dafür lediglich eine Exe- oder Msi-Datei im Silent-Modus gestartet). Ging alles gut, befindet sich die Programmdatei in der Regel im *ProgramData*-Verzeichnis (*C:\ProgramData\chocolatey\lib*) und kann unter Windows 10 über das Startmenü gestartet werden.

▶ **Hinweis** Unter http://chocolatey.org steht ein PowerShell-Modul mit Functions wie *Install-ChocolateyPackage* zur Verfügung, über das ebenfalls ein Package installiert wird.

Es gibt mindestens drei Alternativen, um den *Chocolatey*-Provider hinzuzufügen. Die erste besteht darin, den Provider als Package zu installieren. Um das „Henne, Ei-Paradoxon" aufzulösen, gibt es den *ForceBootstrap*-Parameter bei *Install-Package*. Er sorgt dafür, dass der erforderliche Provider gleich mitinstalliert wird. Damit lässt sich die Installation eines Pakets in einem Aufruf erledigen.

Auch wenn das Installieren von Paketen per *Choco.exe* im Allgemeinen die bessere Alternative ist, da dieses Tool insgesamt besser funktioniert, wird für die folgenden Beispiele das *PackageManagement*-Modul verwendet.

Beispiel

Der folgende Befehl fügt per *Install-Package* das Paket „Autohotkey" hinzu und lädt gleichzeitig den erforderlichen *Chocolatey*-Provider (Abb. 7.3).

```
Install-Package Autohotkey -ProviderName Chocolatey -ForceBootstrap
```

Ging alles gut, befindet sich nicht nur die *Autohotkey*-Installation im Programm-Verzeichnis, es wurden auch der *Chocolatey*-Provider und die *Chocolatey*-Paketquelle installiert, so dass *Find-Package* in Zukunft „ein paar" Pakete mehr findet.

▶ **Hinweis** Neben *Chocolatey* gibt es auch einen Package-Provider mit dem Namen *ChocolateyGet*. Dieser kapselt das Befehlszeilentool *Choco.exe* und ist in erster Linie für das Abrufen von Paketen gedacht.

Eine weitere Alternative stammt aus der Anfangszeit des *PackageManagement*-Projekts und besteht in dem „versteckten" Bootstrap-Provider, der beim Aufruf von *Find-Package* angegeben wird.

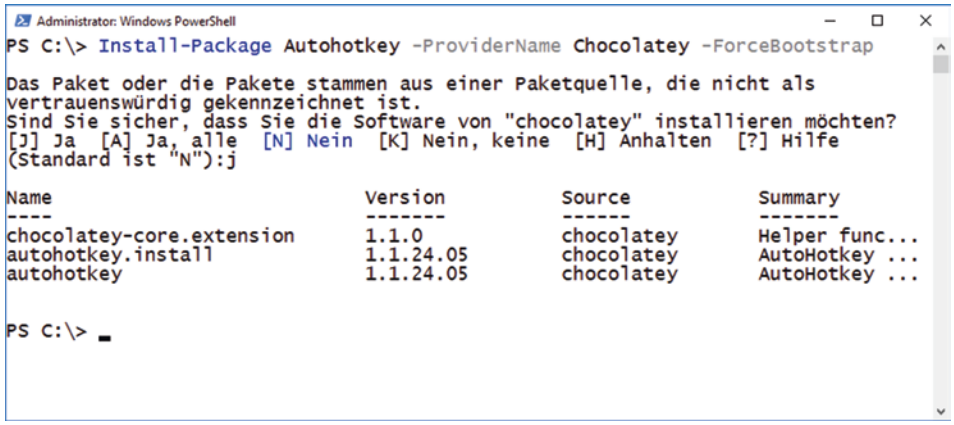

Abb. 7.3 Dank „Bootstrapping“ wird die Paket-Installation sehr einfach

Beispiel

Der folgende Befehl installiert ebenfalls den *Chocolatey*-Provider.

```
Install-Package -Name Chocolatey -ProviderName Bootstrap
```

Die dritte Alternative ist das *Install-PackageProvider*-Cmdlet.

Beispiel

Der folgende Befehl fügt erneut den *Chocolatey*-Provider hinzu.

```
Install-PackageProvider -Name Chocolatey -Force
```

Alle Befehle setzten natürlich eine Internet-Verbindung voraus. Ein bereits heruntergeladenes Paket kann von einem anderen Arbeitsplatz über den *Source*-Parameter von *Install-Package* geladen werden.

▶ **Hinweis** Die Provider-Assembly wird unter *C:\Program Files\PackageManagement\ProviderAssemblies* abgelegt, außerdem wird das Verzeichnis *C:\Chocolatey* angelegt, in dem die heruntergeladenen Pakete abgelegt werden. Ein weiteres Verzeichnis ist der *NuGet*-Cache unter *C:\Users\PsHub-Admin\AppData\Local\NuGet\Cache*. Hier werden alle heruntergeladenen Paketdateien abgelegt.

Der *Force*-Parameter bei *Install-Package* ist erforderlich, da alle Provider und Paketquellen zunächst als „Untrusted“ eingestuft werden; die Eigenschaft *IsTrusted* der Paketquelle besitzt den Wert *$false*. Das lässt sich per *Set-PackageSource*-Cmdlet nachträglich ändern.

Beispiel

Der folgende Befehl setzt die Paketquelle *Chocolatey* auf „Trusted“.

```
Set-PackageSource -Name Chocolatey -IsTrusted
```

▶ **Tipp** Für das Betrachten von *NuGet*-Paketdateien (Erweiterung.*nupkg*) gibt es den *NuGet-Package-Explorer* (Download unter https://npe.codeplex.com/downloads/get/clickOnce/NuGetPackageExplorer.application). Er macht den Inhalt einer Nupkg-Datei sichtbar.

7.1.5 Package-Provider offline installieren

Nicht jeder Computer ist mit dem Internet verbunden oder soll Packages direkt laden. Das Einrichten einer Paketquelle im Intranet ist grundsätzlich kein Problem. Soll jeder Arbeitsplatz darauf zugreifen können, muss beziehungsweise müssen dort die Assembly-Dateien des Package Provider vorhanden sein. Bei WMF 5.0 kann ein Package-Provider nur über das Internet installiert werden. Mit der Version 5.1 wurde bei *Install-PackageProvider* der *Source*-Parameter eingeführt. Über ihn wird ein Verzeichnis ausgewählt, in dem sich die Provider-Dateien befinden. Auch dieses Merkmal scheint noch nicht ganz ausgereift zu sein und setzt voraus, dass die Provider-Dateien als Dll-Datei vorliegen.

7.1.6 Das PowerShellGet-Modul für die Modul- und Skriptverwaltung

Für die Modul- und Skriptverwaltung gibt es das *PowerShellGet*-Modul mit seinen 25 Functions (Tab. 7.3). Im Mittelpunkt steht die Function *Install-Module*, über die Module und DSC-Ressourcen geladen werden. Module werden per *Find-Module* gesucht. Für das Auffinden einer DSC-Ressource gibt es die *Find-DSCResource*-Function. Auch eine DSC-Ressource wird per *Install-Module* hinzugefügt, da sie als Modul vorliegt. *PowerShellGet* ist auch für das Laden von Skripten zuständig. Ein *Find-Script* beziehungsweise *Find-Command* durchsucht Repositorys nach Skripten. Als Suchkriterium steht nicht nur der Name zur Auswahl, sondern auch die Inhalte von Tags („Markierungen"), die als Stichwörter in das Skript beziehungsweise die Manifestdatei eines Moduls eingetragen wurden.

7.1.7 Die ersten Schritte mit PowerShellGet

Bei PowerShellGet gibt es Repositorys (Repos) anstelle von Paketquellen. Repository ist ein eher generischer Begriff. Auch ein Verzeichnis mit Ps1-Dateien lässt sich als Repository bezeichnen. Was ein PowerShellGet-Repository von einem schlichten Verzeichnis unterscheidet sind die Metadaten (die unter anderem eine Versionsnummer, den Namen des Autors und andere Informationen umfassen) und natürlich der Umstand, dass es mit Functions wie *Find-Module* oder *Find-Script* angesprochen werden kann.

▶ **Hinweis** Alle Beispiele in diesem Abschnitt beziehen sich auf das Repository „PoshRepo" des Autors unter der Adresse https://www.myget.org/F/poshrepo/api/v2.

Tab. 7.3 Die wichtigsten Functions aus dem PowerShellGet-Modul im Überblick

Function	Bedeutung
Find-Command	Sucht nach „Commands“ (also Cmdlets, Functions, Alias und Workflows) innerhalb der Module eines Repositorys.
Find-DSCResource	Beschränkt die Suche auf DSC-Ressourcen.
Find-Module	Sucht Module und DSC-Ressourcen.
Find-RoleCapability	Spielt bei JEA („Just Enough Administration“) eine Rolle. Es findet alle „Role Capabilities“, die in der Psrc-Datei definiert ist, die dem Endpoint bei dessen Registrierung zugeordnet wurde.
Find-Script	Gibt alle Skripte zurück. Durchsucht werden entweder alle oder nur das oder die ausgewählten Repositorys.
Get-InstalledModule	Gibt die Module zurück, die per PowerShellGet hinzugefügt wurden.
Install-Module	Fügt ein Modul lokal hinzu. Der Scope-Parameter entscheidet, ob das Modul für alle Benutzer oder nur den aktuellen Benutzer hinzugefügt wird.
New-ScriptFileInfo	Legt die Metadaten für eine Skriptdatei an, so dass diese per Publish-Script in einem Repository veröffentlicht werden kann.
Publish-Module	Veröffentlicht ein Modul, das auf dem Computer vorhanden ist, als NuGet-Package in dem angegebenen Repository. Als Berechtigungsnachweis wird ein API-Key benötigt, über den jeder am Repository registrierte Benutzer verfügt.
Register-PSRepository	Registriert ein Repository.
Save-Module	Speichert ein Modul, das zuvor per Find-Module gefunden wurde.
Set-PSRepository	Ändert eine oder mehrere Einstellungen an einem Repository.
Test-ScriptFileInfo	Testet, ob die Manifestinformation für eine Skriptdatei gültig ist.
Uninstall-Module	Entfernt ein per Install-Module kopiertes Modul wieder.
Update-Module	Aktualisiert ein oder mehrere Module aus einem Repository und bringt sie damit auf den aktuellen Stand.
Unregister-PSRepository	Entfernt ein Repository.

Ein „Get-PSRepository“ listet alle aktuell verfügbaren Repos auf. Mit der PSGallery ist auch diese Liste am Anfang mehr als überschaubar, wenngleich sie vollkommen ausreichend ist. Sollten andere Portale wie die TechNet Script Gallery oder *poshcode.org* eines Tages auch als NuGet-Repository ansprechbar sein, müssten diese über die *Register-PSRepository*-Function eingebunden werden.

Die Functions *Find-Module*, *Find-Script*, *Find-DSCResource* und *Find-Command* geben standardmäßig den Inhalt der *PSGallery* aus.

Beispiel

Der folgende Befehl gibt die Eckdaten zu allen Modulen aus, die unter der *PSGallery* zur Verfügung stehen.

```
Find-Module -Repository PSGallery | Sort -Property Name
```

Beispiel

Der folgende Befehl lädt ein Skript von der PowerShell Gallery.

```
Install-Script -Name GetExtendedFileProperties -Repository PSGallery –
Force
```

Die Ps1-Dateien werden unter *C:\Program Files\WindowsPowerShell\Scripts* abgelegt. Beim ersten Download muss das Hinzufügen des Verzeichnisses zur *Path*-Umgebungsvariablen bestätigt werden. Das Repository muss nur ausgewählt werden, wenn sich die Datei in mehreren Ablagen befinden sollte.

Sollten weitere Repository einbezogen werden, müssen diese per *Register-PSRepository* hinzugefügt werden.

Beispiel

Der folgende Befehl fügt das öffentliche Repository des Autors hinzu.

```
Register-PSRepository –Name PoshRepo -SourceLocation
https://www.myget.org/F/poshrepo/api/v2
```

Damit kann „PoshRepo“ in die Suche einbezogen werden:

```
Find-Module -Repository PoshRepo
```

Ein kleiner Nachteil ist natürlich, dass wenn ein Repository aus irgendeinem Grund nicht erreichbar sein sollte, jede Suche, bei der alle Repositorys einbezogen werden, etwas länger dauert, da immer auch die nicht erreichbaren Repositorys abgefragt werden.

Wie sich Skripte und Module in einem Repository veröffentlichen lassen, wird in Abschn. 7.2.4 gezeigt.

7.2 Eigene Ablagen für Module und Skripte einrichten

Wo lassen sich Module und Skripte am besten ablegen, so dass sie für alle, die sie einsetzen sollen, leicht zugänglich sind? Berücksichtigt man den Umstand, dass es sich bei einem Skript um eine Textdatei und bei einem Modul um eine Gruppe von Dateien handelt, die in einer einfachen Verzeichnishierarchie abgelegt sind, ergeben sich jede Menge Alternativen. Im einfachsten Fall werden die Dateien in einem Web- oder Ftp-Verzeichnis mit individuellen Zugriffsberechtigungen abgelegt. Heruntergeladen werden sie per *wget* (dem Alias für das *Invoke-WebRequest*-Cmdlet).

Beispiel

Der folgende Befehl lädt die Ps1-Dateien des Buches vom Ftp-Verzeichnis des Autors als Zip-Datei herunter und legt die Datei unter *C:\Temp* ab. Benutzername und Kennwort werden als *PSCredential*-Objekt übergeben.

```
$Cred = [PSCredential]::new("ftp12146773-pskurs", ("posh2017" |
ConvertTo-SecureString –AsPlainText -Force)
wget -Uri ftp://wp12146773.server-he.de/Posh/PoshBuchSkripte.zip -
Credential $Cred -OutFile C:\Temp\PoshBuchSkripte.zip
```

Noch praktischer wäre es natürlich, wenn die Datei gleich an die *Expand-Archive*-Function der PowerShell „gepiped" werden könnte. Doch da diesen Komfort weder das Cmdlet noch die Function bietet, ist ein Zwischenschritt erforderlich:

```
wget -Uri ftp://wp12146773.server-he.de/Posh/PoshBuchSkripte.zip -
Credential $Cred -OutFile PS1Skripte.zip;Expand-Archive -Path
.\PoshBuchSkripte.zip -DestinationPath C:\temp –Force
```

Etwas umständlich, aber leider anders nicht machbar.

▶ **Hinweis** Sollte das *Pscx*-Modul vorhanden sein, gibt es bereits ein *Expand-Archive*-Cmdlet, so dass anstelle von *DestinationPath* der Parameter *OutFile* verwendet werden muss. Soll explizit die Function verwendet werden, muss der Modulname vorangestellt werden (*Microsoft.PowerShell.Archive\Expand-Archive*).

Neben den klassischen Web- und Ftp-Verzeichnissen gibt es natürlich noch weitere Alternativen. Wir leben im Zeitalter der Cloud, daher spricht nichts dagegen, Speichorte wie Azure Storage, Amazon S3 oder Dropbox als Ablage zu verwenden. Für diese Dienste gibt es nicht nur eine REST-API, so dass ein Abrufen per *Invoke-RestMethod* möglich ist, sondern im Allgemeinen auch Cmdlets, durch die die REST-API gekapselt wird.

Beispiel

Das folgende Beispiel lädt ein Modul aus einem Azure-Speicherkonto herunter und kopiert die ausgepackte Version in ein Modulverzeichnis, das zuvor angelegt wurde. Für den Download muss zuerst ein Speicherkontext angelegt werden. Sehen Sie das Beispiel bitte nur als Anschauungsbeispiel für die generelle Vorgehensweise. Damit es auch praktisch nachvollziehbar ist, ist natürlich ein Azure-Konto erforderlich. Auch die Namen für Speicherkonto und Blob (die Namen müssen klein geschrieben werden) müssen entsprechend angepasst werden.

```
<#
 .Synopsis
 Datei aus Azure Storage Account laden
 .Notes
 Es kommt auf die Groß-/Kleinschreibung bei den Namen an
#>

$BlobName = "PsKursModul.zip"
$ModulePath = "C:\Program Files\WindowsPowerShell\Modules\$ModulName"
$ContainerName = "ps1blob"
$LocalPath = $env:Temp
$StorageAccountName = "standardspeicher"
$StorageAccountKey =
"6PUTaqeopMk3EDwse7gyNfazXobI0GjK74l7cuaWZ4se7KWcCDuJ3vOg8qAnU9wcmvqnTwhU
uxBWSC47OmofFw=="
$ModulName = "PsKurs"

try
{
  $StorageContext =  New-AzureStorageContext -StorageAccountName
$StorageAccountName `
   -StorageAccountKey $StorageAccountKey -ErrorAction Stop
}
catch
{
  Write-Warning "Fehler beim Anlegen des Speicherkontextes ($_)"
  exit -1
}

try
{
  Get-AzureStorageBlobContent -Blob $BlobName -Container $ContainerName `
   -Context $StorageContext -Destination $LocalPath -Force -ErrorAction
Stop
  Write-Host -ForegroundColor White "Download erfolgreich..." -
BackgroundColor Green
}
catch
{
  Write-Warning "Fehler beim Download des Blobs ($_)"
  exit -2
}

# Jetzt auspacken und in ein Modulverzeichnis kopieren
try
{
  mkdir $ModulePath -ErrorAction Ignore | Out-Null
}
catch
{
  Write-Warning "Fehler beim Anlegen des Modulverzeichnis ($_)"
  exit -3
}

Expand-Archive -Path (Join-Path -Path $env:temp -ChildPath $BlobName) -
DestinationPath $ModulePath
```

7.2.1 Skripte über GitHub als „Gists“ abrufen

Gist ist der Name eines relativen neuen Angebots des Open Source-Portals *GitHub.* Ein Gist ist nichts anderes als eine Datei, die unter *GitHub* abgelegt wurde, und die direkt über eine URL angesprochen werden kann. Ein Beispiel ist die Adresse https://gist.github.com/pemo11/d37ee6376b9cd29558f6687e3e3cd47a, über die ein Skript des Autors abgerufen wird. Dank eines *PackageManagement*-Providers von PowerShell-Experte *Doug Finke* wird der Download solcher Gist-Dateien sehr einfach.

Im ersten Schritt wird der Provider per *Install-PackageProvider* installiert:

```
Install-PackageProvider -Name GistProvider
```

Da der Provider „Gist“ heißt, ist noch einmal ein Import erforderlich:

```
Import-PackageProvider –Name Gist
```

Anschließend listet ein *Find-Package* alle Skripte auf, die über diesen Provider von einem Autor zur Verfügung stehen. Eine generelle Suche ist nicht vorgesehen, da theoretisch eine sehr, sehr große Zahl von Dateien zurückgegeben würde. Der folgende Befehl listet alle Beispielskripte des Autors auf:

```
Find-Package -Source pemo11 –Provider Gist
```

7.2.2 Repository statt Webverzeichnis

So reizvoll der direkte Download von Dateien aufgrund seiner Einfachheit auch sein mag, optimal ist er nicht, da bei dieser Variante Metadaten keine Rolle spielen. Bei theoretisch tausenden von Dateien ist es nicht praktikabel, diese Dateien auf Verzeichnisse zu verteilen oder immer alle Dateien komplett herunterladen zu müssen. Bevor jemand auf die Idee kommt, eine Volltextsuche mit *Lucene.Net* aufzusetzen,[4] die PowerShell hat dafür seit der Version 5.0 eine Lösung. Diese heißt PackageManagement und wurde zu Beginn des Kapitels bereits ausführlich vorgestellt. Die Cmdlets aus dem *PackageManagement*-Modul können nicht nur Module und Skripte aus einem Repository abrufen, sondern diese auch in einem Repository veröffentlichen. Die Grundlage dafür sind *NuGet*-Packages, die per *Publish-Module* und *Publish-Script* erzeugt werden.

Fehlt nur noch ein geeignetes Repository. Mit der *PowerShell Gallery* gibt es bereits eine Option. Doch da alle Module und Skripte öffentlich sind und es zudem auch nicht Sinn und Zweck der *PowerShell Gallery* ist, als Ablage für die eigenen Module und Skripte zweckentfremdet zu werden, kommt diese Option nicht in Frage. Da Microsoft den Quellcode der PowerShell Gallery inzwischen auf GitHub unter https://github.com/PowerShell/PSPrivateGallery („Private PSGallery“) veröffentlicht hat, wäre dies eine

[4] Eine beliebte Technik für eine Volltextsuche in beliebigen Dateiablagen.

interessante Alternative. Allerdings stand das Projekt Stand Mai 2017 noch in der Anfangsphase, so dass es zu diesem Zeitpunkt noch keine echte Alternative war.

Technisch basiert ein Repository auf einem *NuGet*-Server. *NuGet* ist ein Open Source-Projekt, so dass es „jede Menge" Alternativen für das Einrichten eines solchen Dienstes gibt. Die folgende Aufzählung erhebt daher keinen Anspruch auf Vollständigkeit:

- Einen *NuGet*-Server im Intranet betreiben. Eine Anleitung gibt es unter anderem unter http://nugetserver.net.
- Einen *NuGet*-Server unter *Azure* betreiben. Auch dafür gibt es zahlreiche Anleitungen im Internet.
- Den kommerziellen Dienst *MyGet* verwenden.
- Das kommerzielle Tool *ProGet* für das Intranet verwenden.

Jede einzelne Option besitzt natürlich ihre Besonderheiten. Meine Empfehlung stammt nicht von Microsoft. Es sind *ProGet* von *Inedo* für das Einrichten eines Repository im Intranet und *MyGet* für eine cloudbasierte Lösung. Beide Angebote gibt es auch in einer kostenlosen Variante, so dass sie auch für Anwender ohne größeres Softwarebudget in Frage kommen. Bei *ProGet* ist eine lokale Installation erforderlich, die zwar dank eines Assistenten sehr einfach gehalten ist, die aber, da unter einem der IIS konfiguriert und eine SQL Server-Datenbank im Spiel ist, nicht immer im ersten Anlauf gelingen muss. Die mit Abstand einfachste Variante bietet daher *MyGet*. Wie sich in wenigen Schritten ein Repository einrichten lässt, zeige ich im nächsten Abschnitt.

7.2.3 Einrichten eines Modul-Repositorys mit MyGet

MyGet ist ein kommerzielles Portal, das sich allgemein an Entwickler richtet, die eine Paketverwaltung für ihre Projekte im Internet einrichten möchten. Da die Paketverwaltung (natürlich) auf *NuGet* basiert, kommt sie auch für PowerShell-Module und -Skripte in Frage. Auch wenn *MyGet* in erster Linie ein kommerzielles Angebot darstellt, gibt es eine kostenlose Variante.[5] Die wichtigste Einschränkung besteht darin, dass der Feed öffentlich ist und theoretisch jeder die veröffentlichten Module abrufen kann (für das Veröffentlichen ist ein API-Key erforderlich). Weitere Infos gibt es unter http://www.myget.org.

Das Einrichten eines Repository im Rahmen des *MyGet*-Portals dauert nur wenige Minuten.

Um ein Modul per *Publish-Module*-Cmdlet auf *MyGet* veröffentlichen (oder laden) zu können, muss *MyGet* zuerst per *Register-PSRepository* registriert werden. In früheren Versionen gab es in dieser Function ein Bug, der bei Verwendung einer Https-Adresse zu einem Abbruch führte. Das *OneGet*-Team bei Microsoft hatte daher einen Workaround in Gestalt der Function *Register-PSRepositoryFix* veröffentlicht, mit dem das Registrieren

[5] Auch GitHub arbeitet gewinnorientiert und beschäftigt laut Wikipedia knapp 600 Mitarbeiter.

auch per Https funktioniert. Die Wahrscheinlich ist groß, dass Sie diesen Workaround nicht mehr benötigen. Trotzdem wird er für die folgende Umsetzung verwendet.

Die Function *Register-PSRepositoryFix* wird in einem Blog-Eintrag des OneGet-Teams vorgestellt, sie ist in zahlreichen Foren (unter anderem Stackoverflow.com) zu finden und auch Teil der Beispiele zu diesem Buch. Sie registriert eine Repository, indem die Daten des Repository direkt in die zuständige XML-Datei *$env:LocalAppData\Microsoft\Windows\PowerShell\PowerShellGet\PSRepositories.xml* eingetragen werden. In der vom Autor erweiterten Function werden für das Repository auch die Eigenschaften *ScriptSourceLocation* und *ScriptPublishLocation* gesetzt, so dass sich auch Skripte in dem Repository veröffentlichen lassen.

Beispiel

Die folgende Befehlsfolge registriert ein unter *MyGet* angelegtes Repository mit dem Namen „PoshRepo“ mit Hilfe der Function *Register-PSRepositoryFix*.

```
$RepoUrlSource = https://www.myget.org/F/poshrepo/api/v2
Register-PSRepositoryFix -Name PoshRepo -SourceLocation $RepoUrlSource -
InstallationPolicy Trusted -Verbose
```

Wurde das Repository registriert, kann ein beliebiges Modul per *Publish-Module* dorthin veröffentlicht werden.

Beispiel

Der folgende Befehl veröffentlicht das Modul *PsInfo*, das lokal vorliegt, in das *PoshRepo*-Repository. Befindet sich das Modul nicht in einem der Modulverzeichnisse, muss sein Pfad angegeben werden. Der Wert für den Parameter *NuGetApiKey* stammt aus dem *MyGet*-Portal (und müsste durch Ihren API-Key ersetzt werden).

```
Publish-Module -Path .\PsInfo -NuGetApiKey "abcdefgh-d006-410a-b040-
acf131460a2f" -Repository PoshRepo
```

Ging alles gut, kann das Modul ab sofort in jeder PowerShell-Konsole per *Install-Module* lokal hinzugefügt werden. Voraussetzung ist lediglich, dass, wie bereits gezeigt, ein Repository per *Register-PSRepository* beziehungsweise *Register-PSRepositoryFix* registriert wurde.

Auch Skripte lassen sich in einem Repository veröffentlichen. Die zuständige Function heißt *Publish-Script*. Da es bei einer Ps1-Datei keine Manifestdatei und damit keine Metadaten gibt, müssen diese zuvor per *New-ScriptFileInfo* angelegt werden. Mit Hilfe der *Test-ScriptFileInfo*-Function werden die Metadaten überprüft.

Beispiel

Im folgenden Beispiel wird die Ps1-Datei „GetExtendedFileProperties.ps1“ aus den Beispielskripten des Buches im Repository *PoshRepo* veröffentlicht.

Im ersten Schritt werden per *New-ScriptFileInfo* die erforderlichen Metadaten in eine neue (!) Ps1-Datei geschrieben, deren Pfad sich in der Variablen *Ps1MetaPfad* befindet:

```
New-ScriptFileInfo -Path $Ps1MetaPfad `
 -Version 1.0 `
 -Author "P. Monadjemi" `
 -Description "Retrieves the extended file properties" `
 -Tags "File", "Extend File Properties", "Explorer" `
 -Force
```

Im nächsten Schritt wird der Inhalt der Ps1-Datei an das Ende der angelegten Metadatendatei eingefügt. Das kann natürlich auch per *Get-Content* erledigt werden:

```
(Get-Content -Path $Ps1MetaPfad, $Ps1Pfad) | Out-File -FilePath $Ps1Pfad
```

Per *Test-ScriptFileInfo* wird die Datei noch einmal überprüft:

```
Test-ScriptFileInfo -Path $Ps1Pfad
```

Vorausgesetzt, der API-Key ist vorhanden, kann die Skriptdatei damit per *Publish-Script* in einem Repository veröffentlicht werden:

```
$ApiKey = "123456789"
Publish-Script -Path $Ps1Pfad -NuGetApiKey $ApiKey -Repository PoshRepo -
Verbose -Force
```

Ob die vergebenen Tags bei einer Suche mit *Find-Script* und dem *Tag*-Parameter berücksichtigt werden, hängt vom Repository ab.

7.2.4 Skripte und Module in der PowerShell Gallery veröffentlichen

Die *PowerShell Gallery* versteht sich als ein Community-Portal, das allen offen steht. Jeder, der sich dazu berufen fühlt, kann daher dort Module, Skripte und DSC-Ressourcen veröffentlichen. Voraussetzung ist lediglich, dass man sich bei der PowerShell Gallery formlos registriert, um den für den Upload erforderlichen API-Key zu erhalten.

Wie es bereits im letzten Abschnitt am Beispiel von *MyGet* gezeigt wurde, muss eine Ps1-Datei eine Reihe von Metadaten enthalten. Am einfachsten werden diese per *New-ScriptFileInfo* angelegt und am Kopf der Datei eingefügt. Diese Angaben werden im Portal angezeigt und sind alleine deswegen erforderlich, damit potenzielle Interessenten mehr über das Skript erfahren. Bei einem Modul sind die Metadaten bereits in der Psd1-Datei enthalten, hier genügt ein Aufruf von *Publish-Module*.

Beispiel

Der folgende Befehl veröffentlicht die mit Metadaten ausgestattete Ps1-Datei Get-ExtendedFileInfo.ps1 in die PowerShell-Gallery.

```
Publish-Script -Path $Ps1Pfad -NuGetApiKey $APIKey -Repository PSGallery
```

Die Ps1-Datei steht unmittelbar danach in der PowerShell Gallery zur Verfügung und kann von der ganzen Welt heruntergeladen werden (Abb. 7.4).

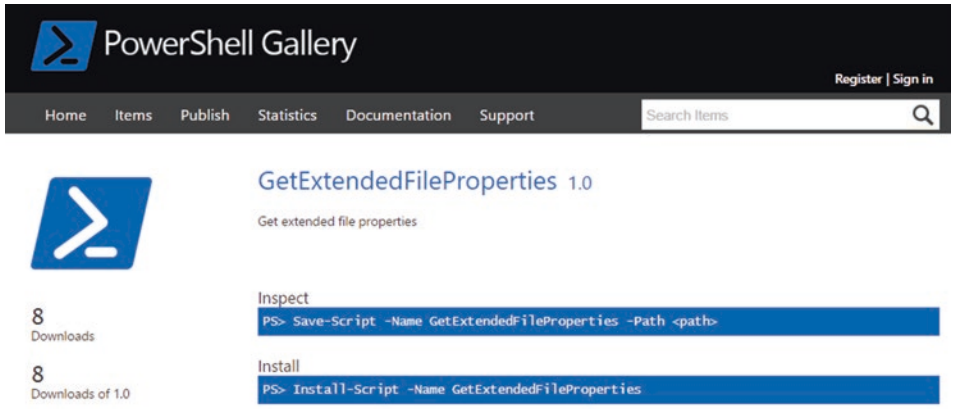

Abb. 7.4 Ein Skript steht unmittelbar nach dem Veröffentlichen in der PowerShell Gallery zur Verfügung

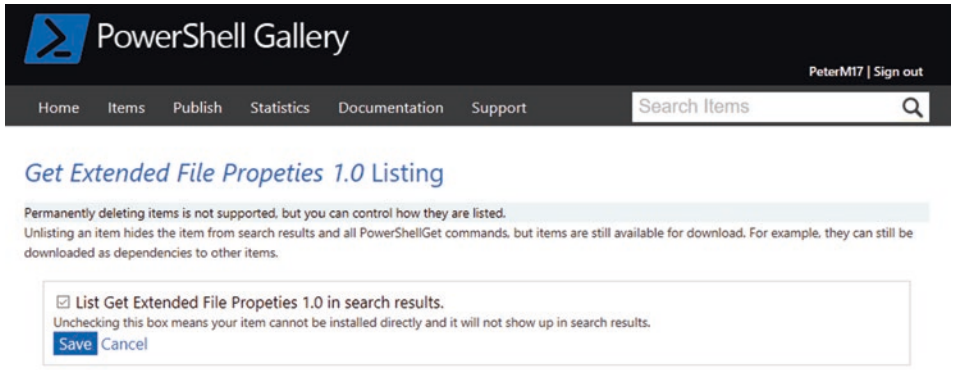

Abb. 7.5 Das Entfernen von Skripten und Modulen aus der PowerShell Gallery ist offiziell nicht vorgesehen

▶ **Hinweis** Wer davon ausgeht, dass jedes von einem anonymen User veröffentlichte Modul oder Skript einem Sicherheitscheck, ähnlich wie bei einem App Store, unterzogen wird, täuscht. Ein Skript steht unmittelbar nach dem Veröffentlichen für alle zum Download zur Verfügung. Man lernt daher den Begriff „Vertrauen" neu zu definieren.[6]

Interessanterweise ist es nicht möglich, ein veröffentlichtes Skript wieder zu löschen (Abb. 7.5). Die Betreiber des Portals wollen damit verhindern, dass es zu Problemen

[6] Zumindest war es im März 2017 so, als der Autor dieses Buches sein Skript „GetExtendedFileProperties.ps1" in der PowerShell Gallery veröffentlichte. Es ist aber davon auszugehen, dass in Zukunft hochgeladene Skripte zumindest per PSScriptAnalyzer untersucht werden, damit sich schlechte Scripting-Techniken nicht global verbreiten.

kommt, falls andere PowerShell-Anwender das Modul oder Skript in ihren Skripten verwenden und es eines Tages nicht mehr zur Verfügung steht. Wer ein Modul oder Skript unbedingt wieder entfernen möchte, muss sich direkt an Microsoft wenden. Ansonsten gibt es lediglich die Möglichkeit, ein Skript so zu entfernen, dass es bei einer Suche nicht berücksichtigt wird.

7.3 Arbeiten mit einer Versionsverwaltung

Sobald Skripte eine gewisse Größe und/oder einen gewissen Funktionsumfang erreichen und im täglichen IT-Betrieb regelmäßig eingesetzt werden, ist eine Versionsverwaltung keine Option mehr, sondern (beinahe) ein Muss. Eine Versionsverwaltung ermöglicht unter anderem, dass mehrere Versionen des Skripts gleichzeitig gepflegt, Änderungen von einem „Projekt-Verwalter" eingepflegt werden können und sich ältere Versionen des Skripts gezielt abrufen lassen. Im einfachsten Fall fungiert die Versionsverwaltung als Back-up-Ablage, aus der die einzelnen Versionen eines Skriptes jederzeit wiederhergestellt werden können. In diesem Abschnitt stelle ich mit *Git* eine populäre Versionsverwaltung vor, die auch für PowerShell-Skripte in Frage kommt. Grundsätzlich kommt natürlich jede Versionsverwaltung für PowerShell-Skripte in Frage, unter anderem auch der *Team Foundation Server* (TFS) von Microsoft, der in der Express Edition kostenlos ist. Für *Git* spricht, dass es schnell und problemlos installiert wird, nur eine minimale Administration erfordert, es eine komfortable Befehlszeile gibt, es als Open Source-Projekt kostenlos ist und es vor allem sehr viel Know-how im Internet und in Gestalt von Videos (unter anderem auf YouTube) und Büchern gibt. Das soll natürlich nicht heißen, dass man innerhalb kurzer Zeit zum *Git*-Experten wird. Trotz einer grundsätzlich einfachen, weil durchdachten Befehlsschnittstelle gibt es zahlreiche Begriffe und Besonderheiten, an die man sich erst langsam gewöhnen wird. Solange es nur um eine Versionsverwaltung für Skripte und Module geht, ist der Einarbeitungsaufwand überschaubar.

▶ **Hinweis** Warum kommt nicht der *Team Foundation Server* (TFS) zum Einsatz? Er stammt von Microsoft, es gibt ihn in der kostenlosen Express Edition, die sich dank eines Assistenten theoretisch in wenigen Schritten installieren lässt, und es gibt Cmdlets, um zum Beispiel einzelne Dateien abzurufen. Aus einem einfachen Grund: Auch wenn der TFS für PowerShell-Skripte grundsätzlich in Frage kommt, für diesen Anwendungsfall wäre er mindestens eine Nummer zu groß.

Im Folgenden werden wir in wenigen Schritten eine auf *Git* basierende Versionsverwaltung in einem Verzeichnis einrichten, ein PowerShell-Modul mit ein paar Dateien anlegen und dieses über einfache *Git*-Kommandos in ein Repo übertragen, das zuvor unter dem *GitHub*-Portal im Internet angelegt wurde. Damit liegt das Modul in einem *GitHub*-Repo und kann von dort auf jeden Computer übertragen werden. Ist es ein privates *Repo*, können

nur autorisierte User das Modul nutzen, ansonsten steht es der ganzen Welt zur Verfügung. *GitHub* ist natürlich nur eine Option. Für das Hosten von Quellcodedateien gibt es attraktive Alternativen wie *Bitbucket*, *GitLab*, *Google Cloud Source* und *Amazon CodeCommit*. *GitHub* ist das größte Portal für Open Source-Projekte und in den vergangenen Jahren zu besonderen Ehren gekommen, da auch Microsoft die wichtigsten Open Source-Projekte, darunter auch die PowerShell-Projekte, unter *GitHub* hostet.

▶ **Hinweis** Sobald *Git* ins Spiel kommt, sollten PowerShell-Skripte nicht mehr mit der ISE, sondern mit Visual Studio erstellt werden, da *Git* (und damit auch *GitHub* beziehungsweise jedes andere Repository im Internet) von Anfang an integriert ist. Mehr zum Zusammenspiel von *Visual Studio Code* und *Git* am Ende dieses Abschnitts.

7.3.1 Schritt für Schritt

Schritt 1: Git for Windows installieren
Im ersten Schritt muss *Git for Windows* installiert werden. Die Downloadadresse ist https://git-for-windows.github.io. Wenn Sie alle Voreinstellungen des Installers übernehmen, sollte anschließend „Git“ über die PowerShell-Konsole oder die Eingabeaufforderung aufrufbar sein (ansonsten müsste der Verzeichnispfad von *Git.exe* nachträglich in die *Path*-Umgebungsvariable für den Computer eingetragen werden).

Schritt 2: Anlegen eines Repos unter GitHub
Der nächste Schritt setzt eine Anmeldung beziehungsweise Registrierung am *GitHub*-Portal unter https://github.com voraus. Das dauert nur ein paar Minuten und ist natürlich kostenlos. Es lassen sich eine beliebige Anzahl an Projekten anlegen. Alle Projekte sind öffentlich, für private Projekte fällt eine monatliche Gebühr an. Für die folgenden Schritte wird ein *GitHub*-Projekt mit dem Namen „PsInfo“ verwendet. Eine *Read.me*-Datei (im Markdown-Format) muss nicht angelegt werden. Sie kann zu einem späteren Zeitpunkt hinzugefügt werden (Abb. 7.6).

Das wichtigste an dem Projekt ist seine Https-Adresse. Diese wird auf der Projektseite so angezeigt, dass sie nicht zu übersehen sein sollte (Abb. 7.7). Außerdem erhalten Sie ein paar Beispiele, wie das Projekt per *Git* von einer Kommandozeile aus angesprochen wird. Für das *PsInfo*-Projekt lautet die Adresse „https://github.com/pemo11/PSInfo.git“.

Git-Kommandos
Git wird über die Befehlszeile gesteuert. Das zentrale Kommando ist „git“, dahinter steckt die Programmdatei *Git.exe* mit einem eigenen Befehlsinterpreter. Auf „git“ folgt immer ein Subkommando. Die Eingabe von „git“ alleine gibt die Syntaxübersicht, ein „git --help“

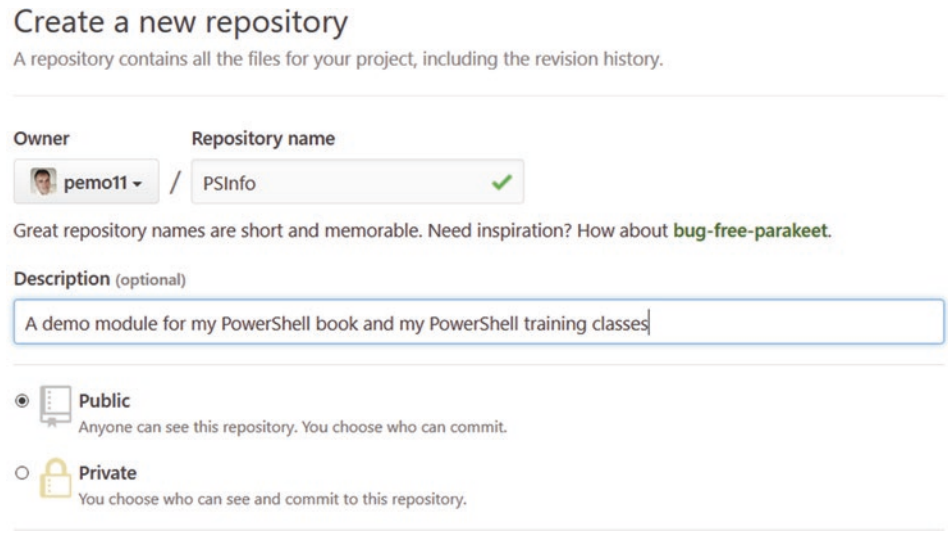

Abb. 7.6 Es wurde ein GitHub-Projekt mit dem Namen „PsInfo“ angelegt

Abb. 7.7 Die Adresse des GitHub-Projekts wird auf der Projektseite angezeigt

gibt alle Subkommandos aus. Wie unter Linux üblich, gehen dem Parameter „help“ zwei Bindestriche (der „double-dash“) voraus. Damit werden allgemein jene Parameter gekennzeichnet, auf die keine weiteren benannten Parameter mehr folgen. Alle noch folgenden Angaben werden als reine Argumente interpretiert. Eine Befehlsfolge kann daher auch mit einem reinen „–“ ohne Parametername abgeschlossen werden. Der Schalter eines Subkommandos, auf das Argumente folgen, beginnt mit einem einfachen Bindestrich.

Schritt 3: Initialisieren eines Verzeichnisses für Git
Das *GitHub*-Portal wird für die nächsten Schritte nicht mehr benötigt. Szenenwechsel zum eigenen Computer. Starten Sie die PowerShell und legen Sie in einem passenden Verzeichnis ein neues Verzeichnis an. Für diese Übung heißt es „PsInfo“. Wechseln Sie in dieses Verzeichnis und initialisieren Sie es für *Git* durch Eingabe von „git init“. Ein „dir“ zeigt nichts an, ein „dir -force“ macht deutlich, dass durch die Initialisierung ein Verzeichnis mit dem unscheinbaren Namen „.git“ angelegt wurde. Dort sind alle Einstellungen und die Informationen über die einzelnen Commits abgelegt.

Schritt 4: Anlegen einer Modulverzeichnisstruktur
Um eine Modulverzeichnisstruktur anzulegen, legen Sie im *PsInfo*-Verzeichnis als erstes ein Verzeichnis mit dem Namen „1.0“ an. Wechseln Sie in dieses Verzeichnis und legen Sie dort eine Psm1- und eine Psd1-Datei mit den Namen „Psinfo.psm1“ und „Psinfo.psd1“ an (Abb. 7.8). Die Psm1-Datei enthält eine Reihe von Functions (in dem Beispiel „Info-Programfiles“, „Info-UnInstallProgs“ „Info-NetVersions“); die Psd1-Datei beschreibt das Modul, legt eine Versionsnummer fest und wählt über den Eintrag *NestedModules* die Psm1-Datei aus:

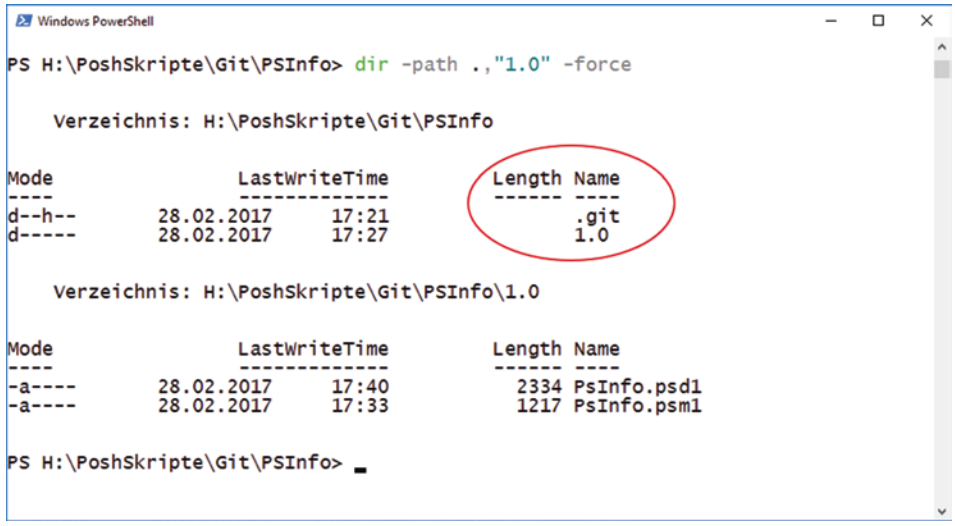

Abb. 7.8 Das PSInfo-Verzeichnis wurde für Git präpariert und enthält aktuell zwei Dateien

```
NestedModules = @('PsInfo.psm1')
```

Auch wenn es nicht zwingend erforderlich ist, werden die Namen der Functions über „FunctionsToExport“ explizit angegeben:

```
FunctionsToExport = @('Info-Programfiles', 'Info-UnInstallProgs')
```

Schritt 5: Dateien zum Staging-Bereich hinzufügen
Im nächsten Schritt wird der Inhalt von *PSInfo* über das *add*-Kommando zur Versionsverwaltung hinzugefügt. Wechseln Sie auf die obere Ebene und geben Sie das folgende Kommando ein:

```
git add *
```

Es fügt alle Dateien (!) auf einmal zur Versionsverwaltung hinzu. Allerdings noch nicht offiziell, sondern nur vorläufig zum „Index“ (ein anderer Ausdruck ist „staging area“). Ein

```
git status
```

gibt aus, dass die beiden Dateien „1.0/PsInfo.psd1“ und „1.0/PsInfo.psm1“ noch nicht „committed“ wurden. Das wird über das *commit*-Subkommando erreicht. Mit dem Schalter *-m* wird ein Kommentar angegeben. Dieser ist mehr als nur ein Formalismus, denn die Kommentare sind ein besseres Unterscheidungsmerkmal als der GUID-Wert, der jedem Commit vergeben wird.

Der folgende Befehl führt ein Commit der hinzugefügten Dateien durch:

```
git commit –m "Mein erstes Commit"
```

Ein erneutes

```
git status
```

gibt nichts mehr aus, da es aktuell keine Dateien im Index gibt. Ein

```
git log --stat
```

gibt die Dateien aus, die aktuell Teil der Versionsverwaltung sind. Um anstelle der GUID den Hashwert des oder der Commit(s) zu sehen, wird der Befehl bereits etwas „komplizierter“:

```
git log --stat --pretty=format:"%h"
```

Per *pretty*-Parameter wird eine Formatierung der Ausgabe durchgeführt. *%h* ist dabei ein Platzhalter für den Hash-Wert.

Schritt 6: Hinzufügen einer weiteren Datei
In diesem Schritt kommt in Gestalt der Datei „Readme.md“ ein Nachzügler hinzu. Legen Sie die Datei zum Beispiel per Ausgabeumleitung im *PsInfo*-Verzeichnis an, sie kann einen beliebigen Inhalt besitzen:

```
"# Mein vielseitiges PSInfo-Modul" > README.md
```

Ein „git status“ gibt an, dass die Datei „Readme.md“ aktuell „untracked“ ist und per „git add“ zum Index hinzugefügt werden muss. Das erledigt der folgende Befehl

```
git add README.md
```

Hat es funktioniert (*git status*)? Wenn nicht, liegt es vermutlich daran, dass es auf die Groß-/Kleinschreibung des Dateinamens ankommt. Ein „git status“ gibt jetzt aus, dass die Datei im aktuellen Zweig („master branch“) entweder committed ist oder das erfolgte Hinzufügen zum Index durch ein „git reset head README.md“ wieder rückgängig gemacht werden könnte.

Wir führen wieder einen Commit mit einem Kommentar durch:

```
git commit -m "Readme-Datei nachgereicht"
```

Ab jetzt wird es etwas interessanter, denn eine Datei, die bereits committed wurde, wird geändert. Konkret: An der Datei „PsInfo.psm1“ wird im Editor eine kleine Änderung vorgenommen. Wie wird die Versionsverwaltung darauf reagieren?

Ein

```
git status
```

macht deutlich, dass die Änderung erkannt wurde. Die Datei „PsInfo.psm1“ wird als „modified“ markiert (Abb. 7.9).

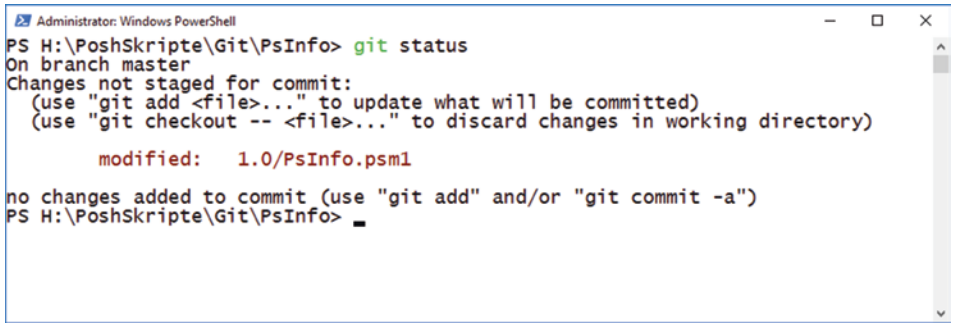

Abb. 7.9 Die an einer Datei vorgenommene Änderung wurde erkannt

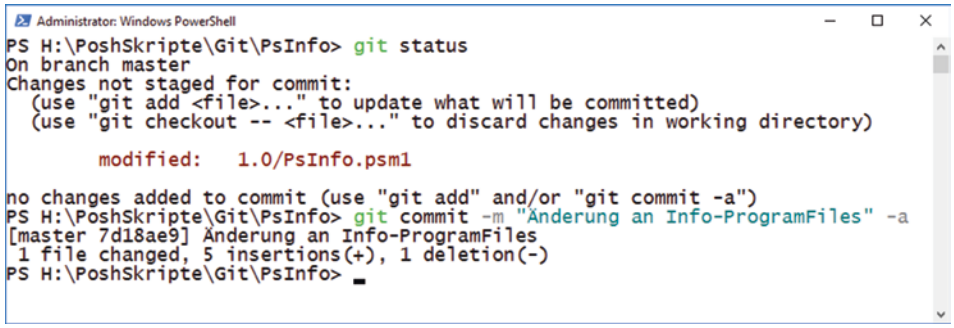

Abb. 7.10 Die geänderte Datei wurde zum Staging-Bereich hinzugefügt und alle Änderungen wurden committed

Damit die Änderungen übernommen werden, muss die Datei erneut zum Zweig hinzugefügt und danach ein Commit durchgeführt werden. Dank dem Schalter *-a* beim *commit*-Subkommando werden beide Schritte in einem Kommando zusammengefasst:

```
git commit -m "Änderung an Info-ProgramFiles" -a
```

Damit ist der Zweig wieder auf dem aktuellen Stand (Abb. 7.10).

Ein

```
git log
```

gibt den bisherigen Verlauf wieder aus. Es wurden zwei Commits durchgeführt, jedem Commit ist eine GUID zugeordnet. In vielen Reporten wird anstelle der GUID lediglich ein kurzer Hashwert ausgegeben.

Schritt 7: Übertragen der Dateien in das GitHub-Repo

Sinn und Zweck einer Versionsverwaltung ist es natürlich nicht, alle Einstellungen einmal ausprobiert zu haben und sich zusätzliche Arbeit aufzuladen. Eine der wichtigsten Daseinsberechtigungen besteht unter anderem darin, in der Lage zu sein, einen älteren Versionstand einer Datei abzurufen. Aktuell liegen alle Versionsstände im aktuellen Verzeichnis vor, in der Praxis legt man eine zentrale Ablage fest, von der aus die Dateien auf einem Computer übertragen werden. Die zentrale Ablage ist das bereits angelegte *GitHub*-Repo.

Bevor die Dateien dahin übertragen werden, soll vorübergehend die an der Datei *PsInfo.psm1* gemachte Änderung wieder rückgängig gemacht werden. Da die Eingabe der vollständigen GUID für die Auswahl eines Commits auf die Dauer etwas umständlich ist, sollten Sie sich zuerst die Hashwerte ausgeben lassen:

```
git log --stat --pretty=format:"%h"
```

Es stellt sich heraus, dass der Commit, der wiederhergestellt werden soll, den Hash „f40d4ad“ besitzt. Das Rückgängigmachen wird durch das *checkout*-Kommando durchgeführt, auf das der Hash des Commits und der Dateiname folgen müssen:

```
git checkout f40d4ad 1.0\PsInfo.psm1
```

Jetzt liegt die Datei wieder in der Version vor der Durchführung der Änderung vor. Diese Version kann erneut editiert und wieder committed werden. Durch einen erneuten Check-out mit dem Hash-Wert des aktuelleren Commits wird die aktuelle Version wiederhergestellt.

▶ **Hinweis** Das Thema „Zurückholen von Dateien aus der Versionsverwaltung“ besitzt naturgemäß viele Facetten. Ein Check-out ist nur eine Option von mehreren.

Im Folgenden soll der aktuelle Stand des Moduls in das angelegte *GitHub*-Repo übertragen werden. Dazu muss dieses Remote-Repo zuerst über das Kommando „remote add“ bekannt gemacht werden:

```
git remote add origin https://github.com/pemo11/PSInfo.git
```

„origin“ ist dabei ein frei wählbarer Name für das Remote-Repo. Im nächsten Schritt werden die lokalen Dateien des „Master“-Branches per *push*-Kommando in das Remote-Repo übertragen:

```
git push –u origin master
```

„Origin“ ist dabei lediglich der lokale Alias für das Remote-Repo, der beim *remote add*-Kommando vergeben wurde. „Master“ ist die Bezeichnung des Hauptzweiges (Abb. 7.11).

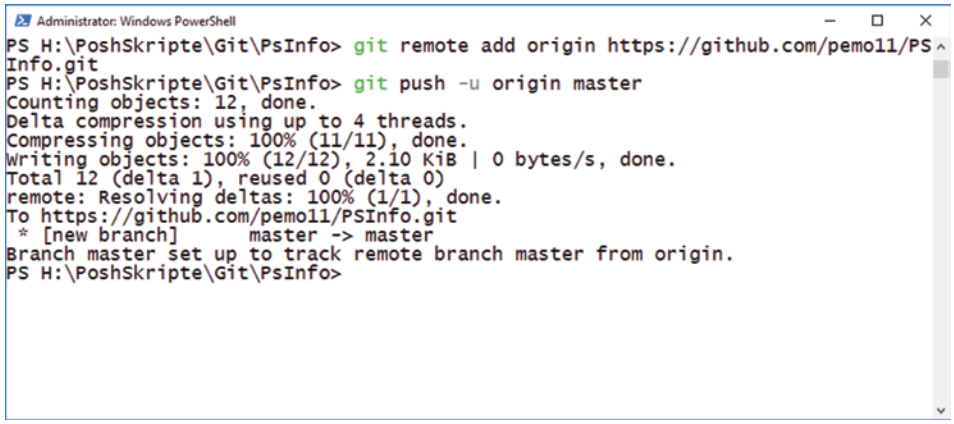

Abb. 7.11 Das lokale Verzeichnis wurde in das Remote-Repo übertragen

Der für die Anmeldung erforderliche Benutzername und das dazugehörige Kennwort werden abgefragt und lokal gespeichert, so dass beim nächsten Push keine Authentifizierung erforderlich ist.

▶ **Hinweis** Sollte das GitHub-Repo eine Readme-Datei enthalten, muss zuerst ein fetch ausgeführt werden, damit das lokale Repo auf den aktuellen Stand gebracht wird. Ein „git fetch origin" wird mit einer Fehlermeldung quittiert werden. Die Ursache ist eine Änderung ab der Version 2.9 von Git für Windows, die den Parameter *--allow-unrelated-histories* erforderlich macht. Der Befehl muss daher „git fetch origin --allow-unrelated-histories" bzw. „git pull origin master --allow-unrelated-histories" lauten.

Ging alles gut, befinden sich die Dateien danach im *GitHub*-Repo und können auf der Webseite des Portals nicht nur betrachtet, sondern auch editiert werden (Abb. 7.12).

Erneuter Szenenwechsel: Anderer Computer, eventuell auch ein anderer Anwender. Der Missionsauftrag lautet: „Hole Dir den aktuellen Stand des *PsInfo*-Moduls aus dem *GitHub*-Repo unter der Adresse https://github.com/pemo11/PSInfo".

Die Lösung könnte einfacher nicht sein. Vorausgesetzt, Git for Windows wurde auf dem anderen Computer installiert, muss lediglich in einem Verzeichnis, in das der Inhalt des Repo-Verzeichnisses übertragen werden soll, das *clone*-Kommando mit der Adresse des *GitHub*-Repo ausgeführt werden:

```
git clone https://github.com/pemo11/psinfo
```

Anschließend liegt ein initialisiertes Git-Verzeichnis mit dem Inhalt des Repos vor. Änderungen werden wieder „committed" und per *push* an das Remote-Repo übertragen.

Damit wären die „Basics" im Umgang mit *Git* beinahe abgehandelt. Ein wichtiger Aspekt steht noch aus. Folgendes Szenario dient dabei als Grundlage: Sie werkeln motiviert vor sich hin, schreiben Zeile um Zeile neue Befehle in das Skript. Am Ende

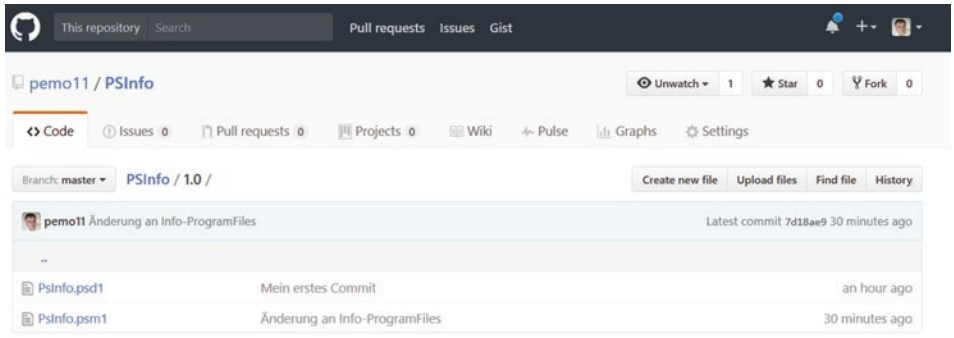

Abb. 7.12 Der Inhalt des lokalen Verzeichnisses liegt im GitHub-Repo vor

funktioniert alles zu ihrer vollsten Zufriedenheit und Sie möchten das Skript in das *GitHub*-Repo „pushen“. Sie führen ein Commit mit dem Schalter *-a* aus und anschließend ein „git push -u origin master“. Doch leider klappt es dieses Mal nicht. Stattdessen ist eine Fehlermeldung die Folge: „Failed to push some refs to https://github.com/pemo11/PSInfo.git“. Mit anderen Worten: Die Versionsverwaltung weigert sich, die Änderungen aufzunehmen. Die Ursache dafür ist in der Regel der Umstand, dass das *GitHub*-Repo seit dem letzten Push verändert wurde und die Online-Version nicht mehr mit dem lokalen Stand, bevor die Änderungen gemacht wurden, übereinstimmt.

Der erste Schritt besteht darin, sich noch einmal zu vergewissern, dass das lokale Repo auf dem neuesten Stand ist und alle Änderungen „committed“ wurden. Der zweite Schritt besteht darin, sich die Unterschiede zwischen der Remote-Datei und der lokalen Datei ausgeben zu lassen. Das erledigt das *diff*-Kommando:

```
git diff origin/master
```

„origin/master“ ist in der Git-Terminologie ein „refspec“, über den das Remote-Repo referenziert wird. Die Änderungen des Remote-Repos sollten in der Konsole ausgegeben werden. Die mit einem + beginnenden grünen Zeilen sind Zeilen, die nachträglich hinzugefügt wurden.

Jetzt geht es darum zu entscheiden, ob die Änderungen in die lokale Kopie übernommen (Merge-Operation) werden sollen oder nicht. Dafür ist das *pull*-Kommando zuständig.

Der folgende Befehl zieht die aktuellen Änderungen aus dem Remote-Repo und fügt sie in die lokale Kopie ein, so dass diese auf dem aktuellen Stand ist:

```
git pull origin master
```

Das Ergebnis ist, wie zu erwarten, ein „Merge Conflict“ in der Datei *PsInfo.psm1*. Doch wo werden die Unterschiede angezeigt? Diese wurden direkt in die Datei eingetragen. Sie müssen jetzt entscheiden, welche Änderungen übernommen werden sollen, indem Sie die Datei entsprechend editieren und danach speichern (Abb. 7.13).

Anschließend werden wieder ein Commit und ein Push in das Remote-Repo durchgeführt:

```
git commit -m "Merge-Konflikte beseitigt" -a
git push origin master
```

Damit sind das lokale Repo und das Remote-Repo hoffentlich wieder auf demselben Stand. Ein erneuter Push hat die Meldung „Everything up-to-date“ zur Folge (Abb. 7.14).

7.3.2 Git-Integration in Visual Studio Code

Wer sich für Git als Versionsverwaltung entscheidet, sollte PowerShell-Skripte mit *Visual Studio Code*, dem Open Source-Allround-Editor von Microsoft, erstellen. Zum einen ist

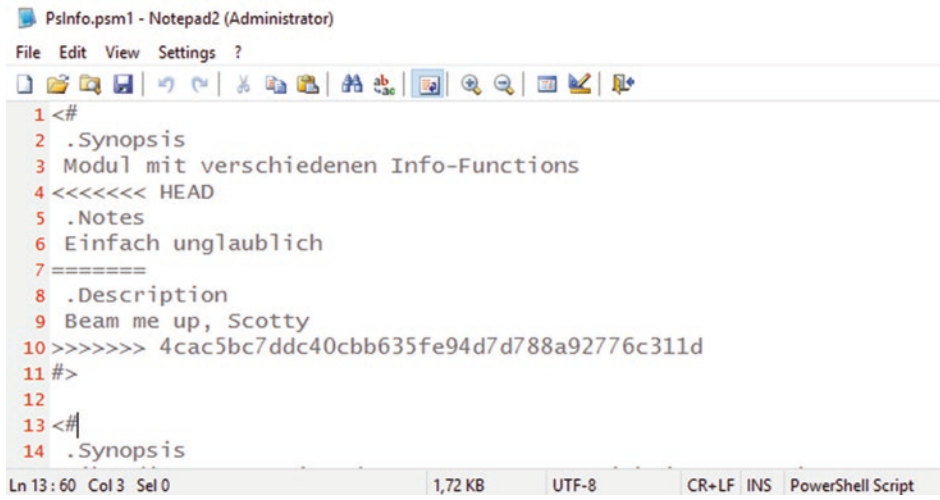

Abb. 7.13 Merge-Konflikte werden direkt in die betroffene Datei eingetragen

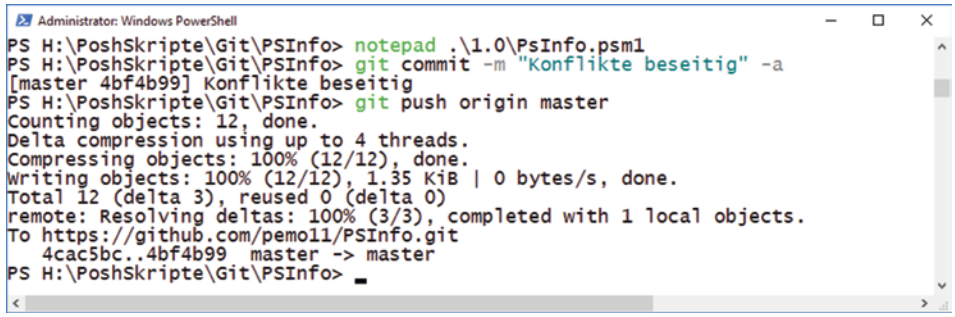

Abb. 7.14 Lokales Repo und Remote-Repo sind wieder auf demselben Stand

Git hier bereits von Anfang an integriert, zum anderen ist Visual Studio Code ein faszinierender Editor, der auch für PowerShell-Skripte sehr gut geeignet ist. Voraussetzung ist natürlich, dass die PowerShell Extensions innerhalb des Editors installiert wurden. Die Git-Integration besteht in einem eigenen Bereich in der Randleiste. Nach dem Öffnen eines Verzeichnisses ist hier der Button **Git-Repository initialisieren** nicht zu übersehen. Über ihn wird ein „git init“ durchgeführt und die Dateien des Verzeichnisses können anschließend durch Eingabe eines Kommentars und einen Klick auf das Häkchen (oder durch Eingabe von [Strg]+[Enter]) „committed“ werden (Abb. 7.15).

Es gibt eine eigene Konsole, in der Git-Ausgaben laufend angezeigt werden. In der Statusleiste wird automatisch der aktuelle Status des Repositorys angezeigt (Abb. 7.16).

Um den Inhalt des ausgewählten Verzeichnisses mit einem Remote-Repository synchronisieren zu können, muss dieses per „git remote add“ zuvor hinzugefügt worden sein:

```
git remote add origin https://github.com/pemo11/PSInfo
```

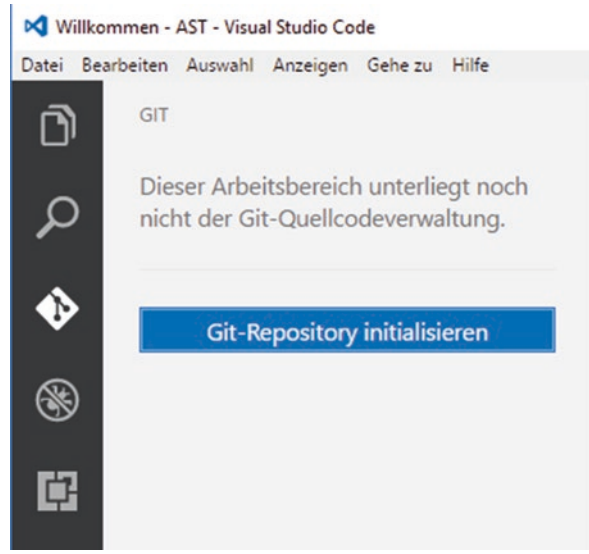

Abb. 7.15 Das aktuelle Verzeichnis kann für Git Repo initialisiert werden

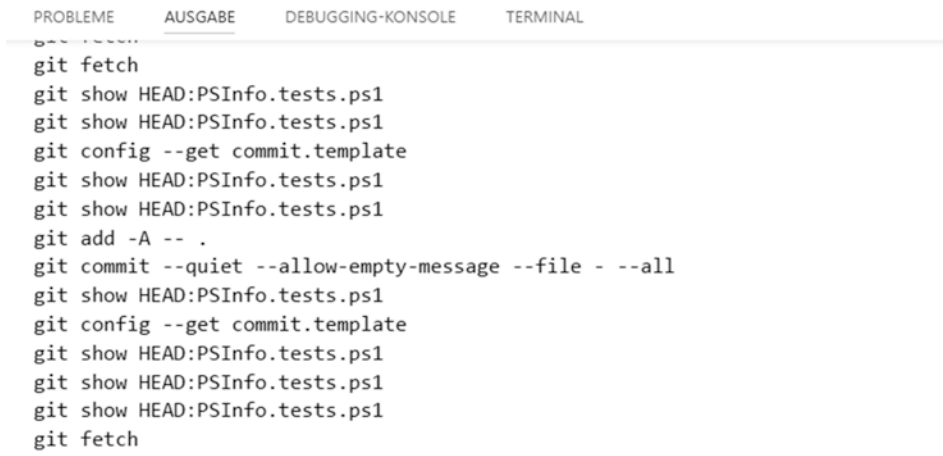

Abb. 7.16 Die Auswirkung jeder Aktion im Editor auf das Git-Repository wird in einem eigenen Bereich der Konsole angezeigt

Wurde eine Remote-Repository erkannt, steht in der Statusleiste rechts vom Namen des aktuellen Zweiges der „Änderungen synchronisieren“-Button zur Verfügung, über den nach einem Commit eine Push-Operation durchgeführt wird (Abb. 7.17).

Beim ersten Mal muss die Option explizit bestätigt werden. Für alle weiteren Aktionen kann die Warnung ausgeblendet werden. Damit ergibt sich beim Arbeiten an einem Skript ein sehr einfacher Workflow, um Änderungen in das Remote-Repository übertragen zu können (Abb. 7.18).

Abb. 7.17 Die Push-Operation wird über die Statusleiste gestartet

Abb. 7.18 Die Push-Operation muss beim ersten Mal explizit bestätigt werden

Die Stärken von Visual Studio Code in Bezug auf die Git-Integration treten vor allem dann auf, wenn vor der nächsten Push-Operation Konflikte gelöst werden müssen und die Dateien im Remote-Repo in der Zwischenzeit verändert wurden. Visual Studio Code stellt die verschiedenen Versionen gegenüber und hebt Unterschiede farblich hervor.

7.4 Aufsetzen einer Release-Pipeline für Module

In diesem Abschnitt stelle ich eine moderne Technik vor, mit deren Hilfe sich Skripte, Module und DSC-Ressourcen so bereitstellen lassen, dass sie allen an sie gestellten Anforderungen genügen und der Autor einen standardisierten Workflow anwendet, der eine Qualitätssicherung beinhaltet. Dazu gehört, dass alle mit *Pester* aufgesetzten Tests bestanden werden und der *PSScriptAnalyzer* (der in Kap. 17 vorgestellt wird) keine Regelverstöße meldet.

7.4.1 Was genau ist eine Release-Pipeline?

Der Begriff „Release-Pipeline“ ist weder ein offizieller Microsoft-Begriff, noch hat er etwas mit der PowerShell-Pipeline zu tun. Der Begriff ist lediglich eine moderne Umschreibung für eine Folge von Schritten, die immer wieder umgesetzt werden muss, damit eine Anwendung bereitgestellt werden kann. Die einzelnen Schritte sind:

1. Quellcode bereitstellen,
2. Build durchführen,
3. Tests und
4. Veröffentlichung.

▶ **Tipp** Ein interessantes Dokument zum Thema Release-Pipeline von Michael Green und Steven Murawski liefert einen sehr guten Einstieg in das Thema: http://aka.ms/thereleasepipelinemodelpdf.

Eine Release-Pipeline ist ein zentraler Bestandteil einer Prozesstechnik, die als „Continuous Integration“, zu Deutsch „Kontinuierliche Integration“, umschrieben wird. In erster Linie bedeutet sie mehr Flexibilität und Agilität für das Bereitstellen von Web-

Anwendungen und fällt in die durch den Begriff „DevOps“ umschriebene neue Art der Herangehensweise bei dem Bereitstellungen von IT-Diensten.

Bei der PowerShell ist natürlich alles eine Nummer kleiner. Da es bei PowerShell keine Anwendungen gibt, geht es hier in erster Linie um das Bereitstellen von Modulen, DSC-Ressourcen und Skripten. Aber auch hier kann es wichtig sein, dass Änderungen an Skripten und Modulen möglichst automatisiert, unkompliziert und vor allem nach der Durchführung einer Qualitätskontrollrolle im Unternehmensnetzwerk bereitgestellt werden.

Frage: Warum sollte man sich mit dem Thema überhaupt beschäftigen? Zunächst bedeutet es mehr Arbeit, mehr Planung, die Auswahl von Alternativen und das Einarbeiten in eine komplett neue Herangehensweise bei einer im Grunde einfachen Aufgabenstellung. Antwort: Weil es ein modernes Thema ist, dass Administratoren und IT-Leuten, die mit dem Bereitstellen von PowerShell-Ressourcen beschäftigt sind, mehr Möglichkeiten und Freiheiten bietet. Insbesondere, wenn mehrere Personen an der Umsetzung eines Moduls oder einer DSC-Ressource mitwirken, stellt eine Release-Pipeline sicher, dass sich ein „Workflow“ einrichten lässt, der sich in der Software-Branche bereits seit vielen Jahren etabliert hat.[7]

In diesem Abschnitt werden mit den Themen Pester-Tests und *PSScriptAnalyzer* zwei Themen vorausgesetzt, die in anderen Kapiteln dieses Buches behandelt werden.

7.4.2 Wo gibt es die Release-Pipeline?

Eine Release-Pipeline ist nicht Teil der PowerShell und auch nicht Bestandteil von Windows Server. Es gibt sie auch aktuell nicht von Microsoft (Stand: März 2017). Microsoft bietet „Continuous Integration“ im Rahmen seiner Azure-Plattform bereits an, es ist aber aktuell auf Webanwendungen beschränkt. Wie in fast allen Bereichen gibt es auch beim Einrichten einer Release-Pipeline mehrere Alternativen. Sie reichen von „Do it yourself“ über die in Mode gekommenen Mischangebote mit einem kostenlosen Einstiegsangebot bis hin zu hochpreisigen Angeboten im Enterprise-Segment für Versicherungen, Banken und andere Branchen mit großem IT-Budget.

Im Folgenden sollen zwei Angebote vorgestellt werden: Die „Do it yourself“-Variante mit Hilfe der Module *PSake*, *PsDeploy*, *Pester* und *PSScriptAnalyzer* und eine komfortable, webbasierte Lösung in Gestalt von *AppVeyor*, einer kommerziellen Plattform mit einem kostenlosen Einstiegsangebot.

▶ **Tipp** Der Übergang von einfachen Skripten zum Bereitstellen von Skripten und Modulen setzt die Einbeziehung einer Versionsverwaltung indirekt voraus. Auch wenn das Veröffentlichen von Dateien in eine Versionsverwaltung wie *Git* direkt in der Befehlszeile möglich ist, etwas mehr Komfort bietet ein Editor, bei dem dies bereits integriert ist. Für die Umsetzung der Übung wird daher *Visual Studio Code* mit der installierten PowerShell-Erweiterung empfohlen, da hier jede Änderung an einer Datei direkt zum Beispiel in ein *GitHub*-Projekt übertragen werden kann (Abb. 7.19).

[7] Außerdem ist es immer gut, über moderne Themen in Grundzügen Bescheid zu wissen.

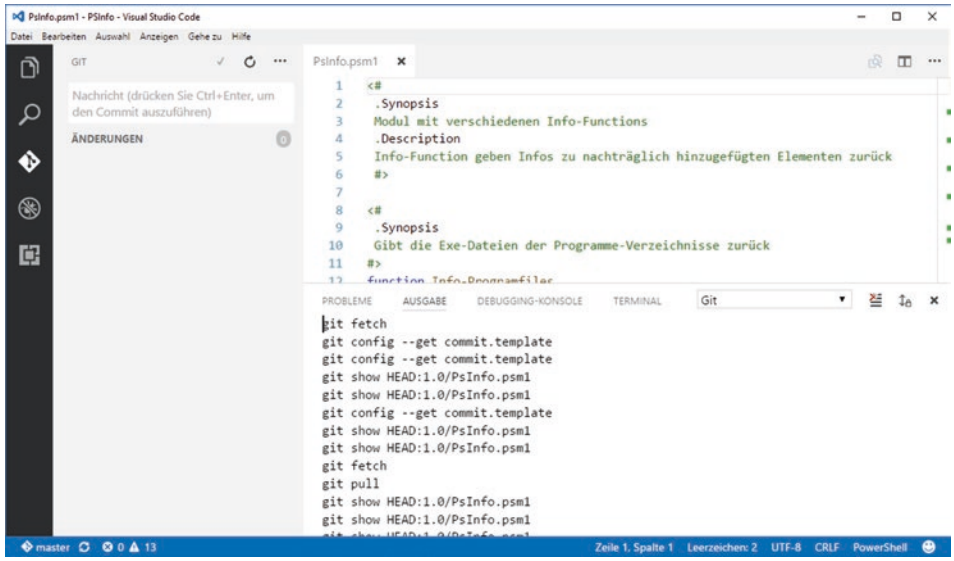

Abb. 7.19 Zu den Vorzügen von VS Code gehört auch eine nahtlose Integration mit Git

7.4.3 Eine Release-Pipeline selber gebaut

In diesem Abschnitt wird eine Release-Pipeline mit Hilfe populärer PowerShell-Module zusammengestellt. Damit es keine Missverständnisse gibt: Eine Release-Pipeline umschreibt lediglich eine Folge von Schritten, die nacheinander ausgeführt werden, um eine Software bereitstellen zu können. Bei einem PowerShell-Skript geht es darum, dass sichergestellt wird, dass es nur dann bereitgestellt wird, wenn es bestimmte Voraussetzungen erfüllt.

Für die Umsetzung sind folgende Skriptdateien involviert:

- ServerInfo.ps
- ServerInfo.tests.ps1
- ServerInfo.psdeploy.ps1
- PSakeBuild.ps1
- StartBuild.ps

▶ **Hinweis** Mit *PSake* und *PSDeploy* werden im Folgenden zwei PowerShell-Module verwendet, die aus Platzgründen in diesem Buch nicht weiter vorgestellt werden. Beide Module werden über die *PSGallery* bezogen. Ihre Projektseiten sind https://github.com/psake/psake und https://github.com/RamblingCookieMonster/PSDeploy.

Ausgangspunkt für die folgende kleine Bereitstellungsübung ist ein sehr einfach aufgebautes Skript mit dem Namen „ServerInfo.ps1“. Es gibt ein paar Details über einen Server in Gestalt eines Objekts zurück:

```
<#
 .Synopsis
 Infos über den Server ausgeben
#>

class ServerInfo
{
  [String]$OsName
  [String]$Architecture
  [DateTime]$LastBootTime
}

[OutputType([ServerInfo])]

$OS = Get-CIMInstance -ClassName Win32_OperatingSystem

New-Object -TypeName ServerInfo -Property @{OSName=$OS.Caption;
Architecture=$OS.OSArchitecture}
```

Das zweite Skript „PSakeBuild.ps1“ ist ein reguläres PowerShell-Skript, das, vergleichbar mit *Pester*, ein paar „Spezialbefehle“ aus dem *PSake*-Modul verwendet, die mehrere Tasks definieren, die nacheinander abgearbeitet werden. Das Besondere ist, dass sich Abhängigkeiten zwischen Tasks festlegen lassen. So lässt sich festlegen, dass wenn Task T1 ausgeführt wird, zuvor die Tasks T2 und T3 ausgeführt werden, da T1 von beiden Tasks abhängig ist. In dem folgenden Skript werden die Tasks „Analyze“, „Test“ und „Deploy“ definiert.

```
<#
 .Synopsis
 Ein PSake-Skript
#>

properties {
    $PsPath = "$PSScriptRoot\ServerInfo.ps1"
}

task Default -Depends Analyze, Test

task Analyze {
    $AResults = Invoke-ScriptAnalyzer -Path $PsPath -Severity @("Error",
"Warning") -Recurse -Verbose:$false

    if ($AResults) {
        $AResults | Format-Table
        Write-Error -Message "Script Analyzer hat Fehler/Warnungen
erzeugt - Build wird abgebrochen."
    }
}

task Test {
  $TResults = Invoke-Pester -Path $PSScriptRoot -PassThru

  if ($TResults.FailedCount -gt 0) {
    $TResults | Format-List
    Write-Error -Message "Ein oder mehrere Tests wurden nicht bestanden -
Build wird abgebrochen."
  }
}

task Deploy -Depends Analyze, Test {
  Invoke-PSDeploy -Path .\ServerInfo.psdeploy.ps1 -Force -Verbose:$false
}
```

Der *Test*-Task ruft ein *Pester*-Skript mit dem Namen „ServerInfo.tests.ps1“ auf:

```
<#
 .Synopsis
 Tests für .\ServerInfo.ps1
#>

describe "Standard-Tests" {

  it "OS-Infos should be something" {
    .\ServerInfo.ps1 | Should Not be $null
  }

  it "LastbootTime should be something" {
    (.\ServerInfo.ps1).LastBootTime -eq [DateTime]::new(0) | Should be
$false
  }

   it "LastbootTime should be something" {
    (.\ServerInfo.ps1).LastBootTime | Should not be ([DateTime]::new(0))
  }
}
```

Der zweite Test schlägt fehl, da in der Klasse die Eigenschaft *LastbootTime* noch keinen Wert besitzt. Damit der Task *Deploy* ausgeführt wird, muss dieser Test ebenfalls bestanden werden.

Das nächste Skript ist „ServerInfo.psdeploy.ps1“. Hier wird mit Hilfe des *PSDeploy*-Moduls festgelegt, dass das getestete Skript in ein anderes Verzeichnis kopiert wird. Für die Praxis bietet *PSDeploy* natürlich noch ein paar andere Optionen wie zum Beispiel das Kopieren in eine VM oder das Veröffentlichen in der *PSGallery*. Wie *Pester* verwendet auch *PSDeploy* eine Dateinamen-Konvention. Per *Invoke-PSDeploy* werden in einem Verzeichnis alle Dateien mit der Erweiterung.*psdeploy.ps1* automatisch ausgeführt.

Das letzte Skript ist „StartBuild.ps1“. Es ruft das *PSake*-Skript auf und startet damit die Bereitstellung.

Das Skript „StartBuild.ps1“ lädt die benötigten Module und startet dann ein *PSake*-Skript:

```
<#
 .Synopsis
 Ein einfaches Beispiel für einen Build-Vorgang, der durch Psake
gesteuert wird
#>

[Cmdletbinding()]
param(
    [String[]]$Task = "Default"
)

if (!(Get-Module -Name Pester -ListAvailable)) { Install-Module -Name
Pester -Scope CurrentUser }
if (!(Get-Module -Name psake -ListAvailable)) { Install-Module -Name
psake -Scope CurrentUser }
if (!(Get-Module -Name PSDeploy -ListAvailable)) { Install-Module -Name
PSDeploy -Scope CurrentUser }

Invoke-Psake -BuildFile .\PSakeBuild.ps1
```

Am Ende wird das *PSake*-Skript per *Invoke-PSake* ohne Angabe eines Tasknamens aufgerufen und damit der Default-Task ausgeführt:

```
Invoke-Psake -BuildFile .\PSakeBuild.ps1
```

Über den *TaskList*-Parameter können Tasks auch direkt aufgerufen werden:

```
Invoke-Psake -BuildFile .\PSakeBuild.ps1 -TaskList Deploy
```

Ging alles gut, wurde die Datei „ServerInfo.ps1“ nach *C:\Temp* kopiert. Wer jetzt einwendet, dass man mit einem simplen Copy-Befehl zum selben Resultat gekommen wäre, übersieht, dass dies nur ein absichtlich einfach gehaltenes Beispiel für eine beliebige Bereitstellungsaktion ist. Der entscheidende Aspekt ist, dass in dem Ablauf nur etwas bereitgestellt wird, dass alle selbst gestellten Anforderungen in Bezug auf die Codequalität und andere Faktoren, die sich zusätzlich im *PSake*-Skript festlegen ließen, erfüllt hat.

7.4.4 Eine Release-Pipeline mit AppVeyor

AppVeyor (http://appveyor.com) ist ein Webportal, auf dem sich .NET-Anwendungen erstellen und bereitstellen lassen. In der Regel liegt der Quellcode der Anwendung in einer Versionsverwaltung (zum Beispiel *GitHub*, *TeamCity* oder *Visual Studio Team Services*). Mit jedem Build-Durchlauf wird eine virtuelle Maschine gestartet, ein Prozess zieht die Quellcodedateien aus der Versionsverwaltung und führt alle Schritte aus, die im Rahmen der Build-Konfiguration festgelegt wurden. Diese wird entweder in der Weboberfläche festgelegt oder basiert auf einer Yaml-Datei (Erweiterung.*yml)*. Der Vorteil von *AppVeyor* ist, dass der Build-Vorgang nicht mehr ausschließlich in einem lokal installierten Visual Studio durchgeführt werden muss und kleine Extras, wie eine Aufbereitung der Testergebnisse oder automatische Benachrichtigungen, hinzugefügt werden können.

Aktuell (Stand: Mai 2017) richtet sich *AppVeyor* ausschließlich an .NET-Entwickler. Der Service ist für Open Source-Projekte kostenlos, ansonsten fällt eine monatliche Gebühr an. Da PowerShell-Skripte keine Anwendungen sind und es keinen Build-Prozess gibt, stellt sich natürlich die Frage, ob *AppVeyor* auch für PowerShell-Anwender in Frage kommt. Die Antwort ist definitiv ja. Wie es in der Einleitung bereits beschrieben wurde, ist eine Release-Pipeline auch bei PowerShell-Ressourcen praktisch. Und genau eine solche Release-Pipeline stellt *AppVeyor* zur Verfügung. Es ist nicht nur am Anfang faszinierend, wenn im Rahmen eines Builds automatisch *Pester*-Tests ausgeführt werden und das Ergebnis als NUnit-XML-Datei gespeichert wird, die anschließend in das *AppVeyor*-Portal geladen wird, so dass alle Testergebnisse im Rahmen der Weboberfläche übersichtlich angezeigt werden.

▶ **Hinweis** Aus Platzgründen und weil es den Rahmen des Buches sprengen würde, kann ich auf den folgenden Seiten *AppVeyor* nur stichwortartig vorstellen. Die verwendeten Beispiele sind Teil der Beispielsammlung, die im

GitHub-Projektportal für das Buch zu finden ist (siehe Einleitung). Nach einer kurzen Einarbeitungszeit kommen auch jene Menschen zum Ziel, die bislang noch keine Berührungspunkte mit dem Thema „Kontinuierliche Integration" haben. Man muss kein Entwickler sein, um *AppVeyor* für das Bereitstellen von PowerShell-Modulen verwenden zu können.

7.4.5 Die ersten Schritte mit AppVeyor

Nach der obligatorischen Registrierung besteht der erste Schritt darin, ein neues Projekt anzulegen. Da der Quellcode aus Repositorys wie *GitHub*, *BitBucket* oder *Visual Studio Team Services* (vormals Visual Studio Online) gezogen wird, muss ein solches Repository für das Projekt ausgewählt werden. Nach der Anmeldung an dem Repository werden alle dort vorhandenen Projekte zur Auswahl angeboten und es wird jenes Projekt ausgewählt, dessen Inhalt vom Build-Prozess verarbeitet werden soll (Abb. 7.20).

Nach dem Anlegen eines *AppVeyor*-Projekts und der Auswahl eines Projekts könnte per „New Build" theoretisch bereits ein erster Build-Lauf mit der Standard-Build-Konfiguration gestartet werden. Das würde aber nicht viel bringen und zu einer Fehlermeldung führen. Über „Settings" muss eine wichtige Einstellung vorgenommen werden. Im Bereich „General" muss festgelegt werden, dass die Einstellungen für den Build-Vorgang aus einer Yaml-Datei stammen, die Teil des Projektportals ist und dort vorher abgelegt wurde. Yaml („Yet Another Markup Language") ist ein relativ neues Textformat, von dem noch die Rede sein wird.

Für die Umsetzung des Beispielprojekts heißt die Yaml-Datei ebenfalls „AppVeyor.yml", wenngleich der Name beliebig ist. Das Häkchen „Ignore appveyor.yml" darf nicht

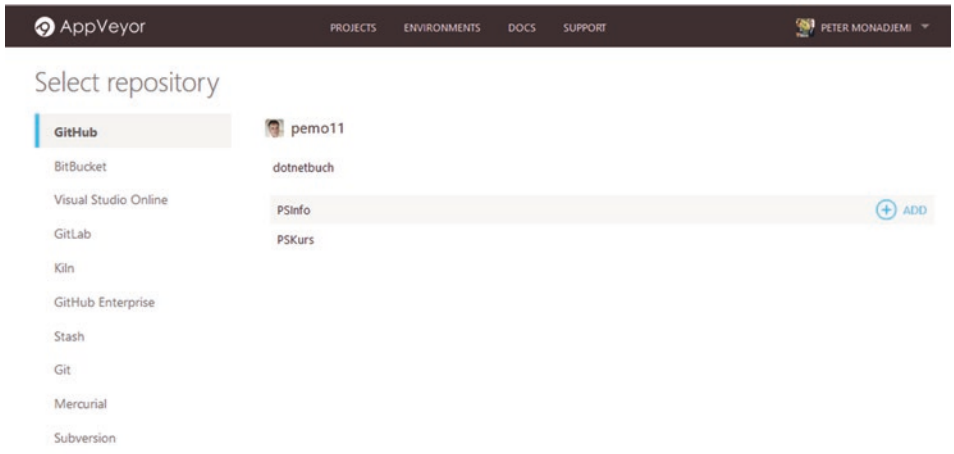

Abb. 7.20 Im ersten Schritt wird ein Projekt angelegt und dabei ein Projekt aus einem Quellcode-Repository ausgewählt

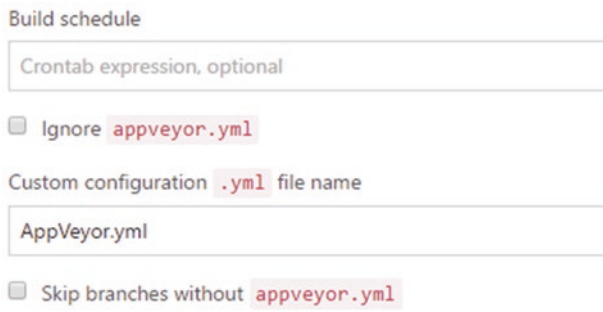

Abb. 7.21 Der Build wird mit einer eigenen Yaml-Datei ausgeführt

gesetzt werden, da ansonsten die Einstellungen innerhalb der Oberfläche maßgeblich sind. Die Yaml-Datei wird nicht in das Portal hochgeladen, sie ist Bestandteil der Versionsverwaltung (Abb. 7.21). Sie wird in Kürze vorgestellt.

Auch wenn unter Settings eine am Anfang etwas verwirrend erscheinende Anzahl an Einstellungen angeboten wird, keine Sorge, die meisten Einstellungen besitzen eine einfache Bedeutung. Im Bereich „General“ werden unter anderem der Projektname, die als nächstes zu vergebene Build-Nummer oder der Zweig in der Versionsverwaltung ausgewählt.

Im Bereich „Environment“ werden unter anderem Umgebungsvariablen definiert, die später bei einem Build-Lauf über die Yaml-Datei abgefragt werden können. Diese Umgebungsvariablen können aber auch direkt in der Yaml-Datei gesetzt werden.

Im Bereich „Notifications“ lassen sich Benachrichtigungen einrichten. Möchte man bei jedem erfolgreichen Build eine E-Mail-Nachricht erhalten, lässt sich das an dieser Stelle einrichten.

Bemerkenswert ist, dass die einzelnen Schritte, die im Rahmen der Benutzeroberfläche konfiguriert werden, jeweils über Cmd- oder PowerShell-Befehle festgelegt werden können.

Für die Bereitstellung eines PowerShell-Moduls spielen diese Einstellungen aber alle keine Rolle, da die komplette Umsetzung für die Yaml-Datei gesteuert wird.

7.4.6 Die Anatomie der Yaml-Datei

Eine Yaml-Datei ist eine Textdatei im *Yaml*-Format. Yaml ist im Vergleich zu XML weniger umfangreich und im Vergleich zu JSON besser lesbar und vor allem leichter editierbar. Manche sehen Yaml als den neuesten Modetrend, ich gehe davon aus, dass in Situationen durchsetzen wird, in denen eine leichte Lesbarkeit wichtig ist. Bei Yaml kommt es auch auf die Einrückungen an. Enthält eine Zeile keine Markierung, muss sie mit zwei Leerzeichen eingerückt werden. Dabei spielen auch Kleinigkeiten eine Rolle. Enthält eine Zeichenkette in einem PowerShell-Skript einen Doppelpunkt, kommt es zu einem Parsing-Fehler. Mehr zu Yaml unter http://yaml.org, eine Gegenüberstellung mit JSON gibt es unter http://yaml.org/spec/1.2/spec.html#id2759572.

7.4.7 Ein PowerShell-Modul per AppVeyor bereitstellen

Im Folgenden soll mit *AppVeyor* ein kleines PowerShell-Modul mit dem Namen „PsInfo“ bereitgestellt werden. Die Release-Pipeline besteht aus mehreren Stufen:

1. Im ersten Schritt werden alle beteiligten Dateien aus dem GitHub-Projektportal kopiert. Dies geschieht automatisch. Sie stehen danach im Verzeichnis *C:\Projects* der AppVeyor-VM zur Verfügung. Alle Pfade, die von den an der Umsetzung beteiligten Skripten verwendet werden, beziehen sich auf dieses Verzeichnis.
2. Im zweiten Schritt werden die benötigten Module (*Pester* und *PSScriptAnlayzer*) und der NuGet-Package-Provider hinzugefügt.
3. Im dritten Schritt wird mit Rahmen eines Testskripts eine Reihe von Pester-Tests ausgeführt.
4. Im vierten Schritt wird der Script Analyzer aufgerufen, um auf die Psm1-Datei einen Satz von Standardregeln anzuwenden.
5. Im letzten Schritt wird eine Modulmanifestdatei angelegt, es wird das PoshRepo-Repository (Wget) eingerichtet und das Modul dorthin veröffentlicht.

Die Yaml-Datei ist sehr einfach aufgebaut, da die Aktionen für die einzelnen Abschnitte in PowerShell-Skripte ausgelagert werden:

```
#      environment configuration  #
version: 1.0.{build}
os: WMF 5

install:
  - ps: . .\AppVeyorSkripts\PsInfoInstall.ps1

environment:
  MySecureVar:
  MyNonSecureVar: NonSecure
  ModuleName: SimpleMath
  ModuleVersion: 1.0

#      build configuration        #
build_script:
  - ps: . .\AppVeyorSkripts\PSInfoBuild.ps1

#      test configuration         #
test_script:
  - ps: . .\AppVeyorSkripts\PSInfoTest.ps1

#      deployment configuration   #
deploy_script:
  - ps: . .\AppVeyorSkripts\PSInfoDeploy.ps1
```

Wie wird die Release-Pipeline in Gang gesetzt? Entweder über den Button „Start new Build“ im allgemeinen Projektportal beziehungsweise „New Build“ auf der Projektseite oder automatisch dadurch, dass die Quelldateien zum Beispiel im *GitHub*-Portal aktualisiert werden. Der Ablauf des Build-Vorgangs wird in einer typischen Konsolenausgabe protokolliert. Hier erscheinen auch die *Write-Host*-Ausgaben der einzelnen PowerShell-Skripte (Abb. 7.22).

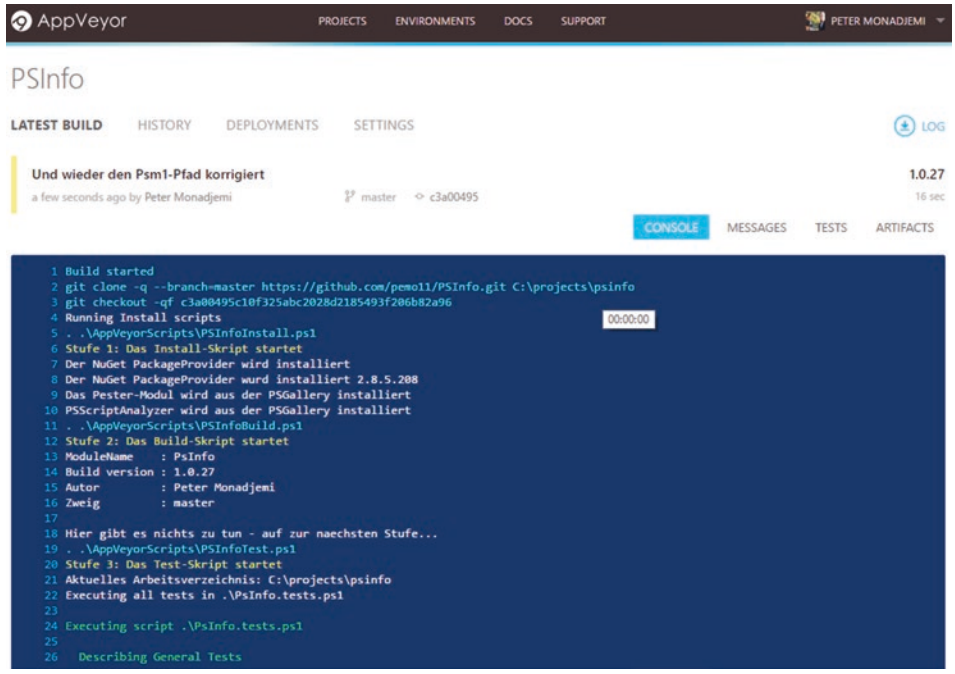

Abb. 7.22 Der Verlauf des Build-Vorgangs wird in der AppVeyor-Konsole ausführlich dokumentiert

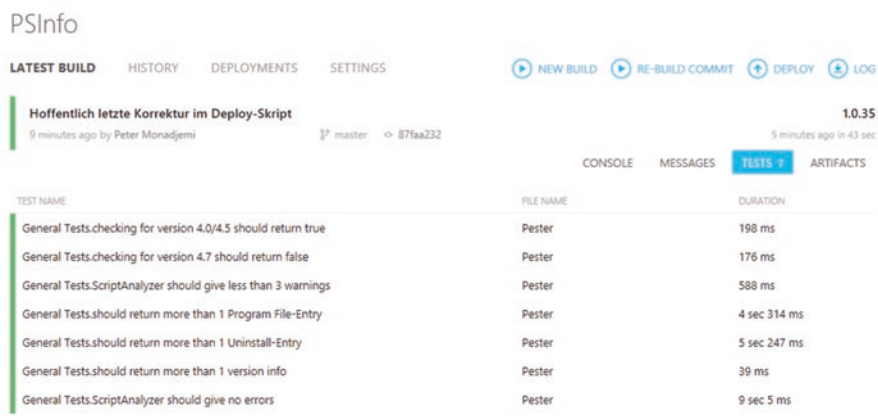

Abb. 7.23 Die Ergebnisse der Pester-Tests werden im AppVeyor-Portal zusammengefasst

Interessant ist der Bereich „Tests“, denn hier werden die Ergebnisse der Pester-Tests zusammengefasst. Dies wird dadurch erreicht, dass die aus den Tests hervorgegangene XML-Datei wieder in das Portal geladen wird (Abb. 7.23).

Es ist am Anfang einiges an „Feintuning“ erforderlich, bis ein Build endlich fehlerfrei durchlaufen wird. Es sind eher Kleinigkeiten wie falsche relative Pfade oder der Umstand, dass ein Modul erst dann über seinen Namen veröffentlicht werden kann, wenn der Modulverzeichnispfad zur *PSModulePath*-Umgebungsvariablen hinzugefügt wurde.

Am Ende steht im Repository unter *MyGet* das Modul in seiner neuesten Fassung, getestet und zum Abruf bereit. Kommen die Dateien aus dem GitHub-Portal, lässt sich im Bereich „Notifications“ ein sogenannter Pull Request einrichten. Damit erhält der Projektverantwortliche eine Benachrichtigung, dass Quellcode vorhanden ist, der in das Projekt eingepflegt werden soll.

7.5 Zusammenfassung

Möchte man Module und Skripte zentral ablegen, so dass sie sich von jedem Arbeitsplatz auf einfache Weise hinzufügen lassen, gibt es dafür mehrere Möglichkeiten. Im einfachsten Fall werden die Dateien über Wget direkt aus einem Webverzeichnis heruntergeladen. Sollen Metadaten wie die Versionsnummer eine Rolle spielen, sollte das Modul per *Publish-Module* in einem Repository abgelegt werden. Dieses kann als Nuget-Server auf verschiedene Weisen angelegt werden. Eine Option ist der kommerzielle Dienst *MyGet*.

Die einzelnen Schritte von der Umsetzung von Quellcode über die Bereitstellung der erstellten Anwendung, zum Beispiel auf einen Webserver, werden unter dem Begriff „Release-Pipeline“ zusammengefasst. *AppVeyor* ist ein Webportal, auf dem sich beliebige auf dem .NET Framework basierende Projekte umsetzen lassen. Bei PowerShell-Skripten gibt es keinen Build-Vorgang. Trotzdem spielt das Thema Release-Pipeline auch hier eine Rolle, wenn Schritte wie Tests oder die Analyse per *PSScriptAnalyzer* einbezogen werden. Eine per *AppYeyor* aufgesetzte Release-Pipeline bewirkt, dass die sich in einer Versionsverwaltung wie *GitHub* befindlichen Skriptdateien automatisiert getestet, analysiert und in einem Repository wie der PowerShell Gallery bereitgestellt werden. Verwendet man für das Erstellen und Bearbeiten der Skripte *Visual Studio Code*, *PowerShell Studio* von *Sapien Technologies* oder einen anderen Editor mit integrierter Versionsverwaltung, kann der Bereitstellungsprozess vollständig automatisiert werden.

DSC-Grundlagen

8

Zusammenfassung

In dieser Lektion stelle ich die „Desired State Configuration“ (DSC) vor. DSC, das bereits mit WMF 4.0 eingeführt wurde, verspricht nicht weniger als eine vollkommen neue und deutlich einfachere Form der Server-Konfiguration im Vergleich zur bisherigen Methode mit Hilfe von Skripten. Anstatt jedes Konfigurationsdetail per Skriptbefehl festlegen zu müssen, wird die Wunschkonfiguration mit Hilfe einer eigens dafür entworfenen Beschreibungssprache festgelegt und danach auf eine beliebige Anzahl an Servern übertragen. Ein Umstand, der DSC sehr attraktiv macht, ist die nahtlose Integration in die PowerShell, so dass das Erlernen von DSC nicht das Erlernen neuer Techniken und Werkzeuge voraussetzt.

8.1 Ein erstes Beispiel

Bevor ich in der gebotenen Kürze auf die Architektur von DSC eingehe und ein paar Worte darüber verliere, warum dem PowerShell-Team so etwas wie DSC überhaupt eingefallen ist, soll ein erstes Beispiel deutlich machen, wie Sie sich DSC vorstellen müssen und wie einfach es ist, DSC selber auszuprobieren. Auch wenn theoretisch ein Windows-PC mit der WMF 4.0 genügt, sollte DSC mit dieser Version nicht mehr verwendet werden – nicht nur wegen der Tatsache, dass mit den Versionen 5.0 und 5.1 wichtige Verbesserungen eingebaut wurden, erst ab der Version 5.0 bietet die ISE auch Eingabehilfen für DSC-Ressourcen an. Eine Alternative ist Visual Studio Code, das mit installierter PowerShell Extension ebenfalls Eingabehilfen anbietet.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_8

Bevor es losgeht, noch ein Hinweis in eigener Sache. Erwarten Sie bitte im Folgenden kein Beispiel, dass zum Beispiel eine ausfallsichere Webserver-Konfiguration mit Lastenausgleich für den Betrieb einer Webanwendung aufsetzt. Im Folgenden geht es, wie könnte es auch anders sein, um ein typisches „Hallo, Welt"-Beispiel. Das Ziel soll es sein, dass auf einem beliebigen Computer ein Verzeichnis existiert und in diesem Verzeichnis eine Datei mit einem bestimmten Inhalt angelegt wird. Das Verzeichnis soll im Stammverzeichnis auf Laufwerk C: angelegt werden und „PoshSkripte" heißen; die Datei soll ganz profan „Test.ps1" heißen und als Inhalt einen Befehl enthalten, der die aktuelle Uhrzeit ausgibt. Als krönender Abschluss soll für das Verzeichnis eine Freigabe angelegt werden. Daran wird deutlich werden, dass auch bei DSC in der Praxis kleinere Probleme auftreten, für die es nicht immer eine einfache Lösung gibt.

Bevor jemand auf die Idee kommen sollte, das Ganze mit *Invoke-Command*, *Set-Content* und einem *Net Use* für das Anlegen der Freigabe anzugehen, soll im folgenden Beispiel diese kleine Aufgabe per DSC gelöst werden.

Der erste Schritt besteht darin, mit dem Befehlswort *Configuration* eine DSC-Konfiguration mit dem Namen „HalloDSC" zu definieren. Eine Konfiguration ist ein mit der Version 4.0 eingeführter Befehlstyp, vergleichbar mit einer Funktionsdefinition. Der Name der Konfiguration kann natürlich frei gewählt werden:

```
configuration HalloDSC
{

}
```

Im nächsten Schritt erhält die Konfiguration „HalloDSC" einen Inhalt. Dieser besteht aber nicht aus PowerShell-Befehlen, sondern aus einer DSC-Ressource. Eine solche Ressource nimmt die Einstellung für ein bestimmtes „Element" vor, das auf dem Zielsystem konfiguriert werden soll. In diesem Fall ist es ein Verzeichnis.

▶ **Tipp** Wer bereits neugierig ist und herausfinden möchte, welche DSC-Ressourcen insgesamt zur Auswahl stehen: Alle verfügbaren DSC-Ressourcen werden über das *Get-DSCResource*-Cmdlet aufgelistet. Am Anfang verfügt auch die PowerShell 5.1 nur über knapp ein Dutzend Ressourcen, die im nächsten Kapitel vorgestellt werden. Weitere Ressourcen werden als reguläres Modul am einfachsten per *Install-Module*-Cmdlet über die PowerShell Gallery geladen. Eine Suche per *Find-Module* nach Modulen, deren Name mit „x" beginnt, gibt bereits einen ersten Überblick über die zur Verfügung stehende Auswahl.

Da auf dem Zielsystem im ersten Schritt ein Verzeichnis angelegt werden soll, könnte die Ressource „Directory" heißen. Ihr Name ist aber stattdessen „File". Auf den Namen der Ressource folgt wieder ein beliebiger Name. Innerhalb der Ressourcen-Definition werden verschiedene Eigenschaften festgelegt, die in der ISE per [Strg]+[Leertaste] angezeigt werden.

Obligatorisch ist die Eigenschaft *Ensure*. Sie erhält den Wert „Present“, wenn die Ressource auf dem Zielcomputer angelegt und „Absent“, wenn die Ressource entfernt werden soll. Über die Eigenschaft *DestinationPath* wird der Verzeichnispfad des anzulegenden Verzeichnisses festgelegt, über die Eigenschaft *Type*, ob eine Datei oder ein Verzeichnis angelegt werden soll. Es gibt weitere Eigenschaften, die im Moment aber keine Rolle spielen.

```
configuration HalloDSC
{

  file TestDir
  {
    Ensure = "Absent"
    DestinationPath = "C:\PoshSkripte"
    Type = "Directory"
  }
}
```

Damit befindet sich die Konfiguration „HalloDSC“ in einem Zustand, in dem sie bereits ausgeführt werden könnte. Doch wie genau wird sie ausgeführt? Ganz einfach durch Eingabe ihres Namens, also „HalloDSC“.

Der folgende Aufruf setzt die Konfiguration „HalloDSC“ um:

```
HalloDSC
```

Das Ausführen der Konfiguration bedeutet nicht, dass die definierten Einstellungen bereits angewendet werden. Es bedeutet lediglich, dass eine Datei mit dem Namen „Localhost.mof“ im Verzeichnis „HalloDSC“ angelegt wird. Die Mof-Datei enthält den gewünschten Konfigurationszustand in einem standardisierten Textformat, dem „Microsoft Object Format“, das auch bei WMI eine Rolle spielt. Der Name „Localhost“ ergibt sich aus dem Umstand, dass in der Konfiguration kein Computername explizit angegeben wurde. Der Name des Verzeichnisses ist „HalloDSC“, da dies der Name der Konfiguration ist.

Wer möchte, kann die Konfiguration bereits jetzt auf den lokalen Computer anwenden, um das Verzeichnis anzulegen. Das geschieht mit dem *Start-DscConfiguration*-Cmdlet, auf das lediglich der Verzeichnispfad folgt, in dem sich die Mof-Datei befindet. Am Anfang sollten immer die Parameter *Wait* und *Verbose* gesetzt werden, da die Konfigurationsänderung durch einen Job (vom Typ „ConfigurationJob“) durchgeführt wird und per *Wait* auf die Beendigung dieses Jobs gewartet wird. Der *Verbose*-Parameter sorgt dafür, dass alle Schritte des *Local Configuration Managers* angezeigt werden.

► **Hinweis** Sollte eine Konfiguration beim letzten Durchlauf vorzeitig abgebrochen worden beziehungsweise noch nicht abgeschlossen sein, muss beim erneuten Aufruf der *Force*-Parameter gesetzt werden, um die unvollständige Konfiguration zu überschreiben.

Der folgende Befehl führt den aktuellen Stand der Konfiguration „HalloDSC“ aus, indem die Mof-Datei im Unterverzeichnis „HalloDSC“ auf den „Zielcomputer“ übertragen und dort vom *Local Configuration Manager* (LCM) angewendet wird:

```
Start-DscConfiguration Path HalloDSC -Verbose –Wait
```

Ging alles gut, wird der Aufruf viel „Output“ produzieren. Außerdem sollte ein Verzeichnis mit dem Namen „PoshSkripte“ auf Laufwerk C: angelegt worden sein. Wird in der *File*-Ressource die Eigenschaft *Ensure* auf den Wert „Absent“ gesetzt, wird das Verzeichnis nach dem erneuten Ausführen der Konfiguration per *Start-DSCConfiguration* wieder entfernt. Eine Konfigurationsänderung lässt sich per DSC auch rückgängig machen beziehungsweise sie kann eine Einstellung auf dem Zielcomputer auch entfernen.

Im nächsten Schritt soll eine Datei im Verzeichnis „PoshSkripte“ angelegt werden. Auch das wird mit der *File*-Ressource bewerkstelligt. Neu ist die Eigenschaft *Contents*, über die der Inhalt der Datei festgelegt wird. Da dieser Inhalt ein PowerShell-Befehl ist mit Anführungszeichen und einem *$*-Zeichen, müssen diese Zeichen als Teil einer Zeichenkette escaped werden:

```
configuration HalloDSC
{
   # Anlegen eines Verzeichnisses
  File TestDir
  {
    Ensure = "Present"
    DestinationPath = "C:\PoshSkripte"
    Type = "Directory"
   }
  # Anlegen einer Datei
   File TestFile
   {
        Ensure = "Present"
        DestinationPath = "C:\PoshSkripte\Test.ps1"
        Type = "File"
        Contents = "`"Die aktuelle Uhrzeit: `$(Get-Date -Format t)`""
    }
 }
```

Mit dem erneuten Ausführen der Konfiguration und dem anschließenden Ausführen von *Start-DSCConfiguration* wird in dem Verzeichnis eine Datei „Test.ps1“ angelegt, die bei ihrer Ausführung die aktuelle Uhrzeit ausgibt.

Bislang wurde die DSC-Konfiguration immer auf dem lokalen Computer angewendet. Das ist jedoch nicht der Sinn und Zweck von DSC. Eine Konfiguration soll auf theoretisch unendlich vielen Computern angewendet werden können. Damit das möglich ist, muss die Konfiguration lediglich um ein *node*-Element erweitert werden. Es ist ein

Pflichtbestandteil einer Konfiguration. Es wurde bislang nur weggelassen, da das erste Beispiel möglichst einfach gehalten werden sollte.

```
configuration HalloDSC
{
  node Server1
  {
    # Anlegen einer Datei
    File TestDir
    {
       Ensure = "Present"
       DestinationPath = "C:\PoshSkripte"
       Type = "Directory"
    }
    # Anlegen eines Verzeichnisses
    File TestFile
    {
      Ensure = "Present"
      DestinationPath = "C:\PoshSkripte\Test.ps1"
      Type = "File"
      Contents = "`"Die aktuelle Uhrzeit: `$(Get-Date -Format t)`""
    }
  }
}
```

Wird die Konfiguration erneut ausgeführt, wird eine Mof-Datei mit dem Namen „Server1.mof“ angelegt. Der Name der Mof-Datei richtet sich nach dem Namen des Nodes. Auch diese Konfiguration wird per *Start-DSCConfiguration*-Cmdlet ausgeführt. Die Voraussetzungen sind natürlich, dass der angegebene Computer existiert, dass er im Netzwerk erreichbar ist, dass die WMF ab Version 4.0 installiert ist und auf dem Computer CIM-Remoting aktiviert wurde. Handelt es sich um den lokalen Computer, wird die Konfiguration auf exakt dieselbe Weise umgesetzt wie bei der Ausführung ohne das *node*-Element.

- **Tipp** Ändert sich der Name des Nodes, wird durch das Ausführen der Konfiguration eine weitere Mof-Datei angelegt. Die bereits in dem Verzeichnis vorhandenen Mof-Dateien werden aber nicht gelöscht, auch wenn sie fehlerhaft sind. Das hat unschöne Fehlermeldungen zur Folge. Da beim Ausführen von *Start-DSCConfiguration* immer alle Mof-Dateien umgesetzt werden, sollten Sie vor dem Ausführen der Konfiguration alle Mof-Dateien in dem Verzeichnis löschen.

In der Praxis wird man auf Node keinen Namen, sondern eine Variable folgen lassen, für die ein Parameter auf die vertraute Art und Weise definiert wird. Auf diese Weise werden der oder die Namen der Node-Computer erst beim Ausführen der Konfiguration festgelegt:

```
<#
  .Synopsis
  Ein Hallo, Welt-Beispiel für DSC
 #>

configuration HalloDSC
{
  param([String[]]$Computername)
  Import-DscResource -ModuleName PSDesiredStateConfiguration

  node $Computername
  {
    # Anlegen einer Datei
    File TestDir
    {
      Ensure = "Present"
      DestinationPath = "C:\PoshSkripte"
      Type = "Directory"
    }
    # Anlegen eines Verzeichnisses
    File TestFile
    {
      Ensure = "Present"
      DestinationPath = "C:\PoshSkripte\Test.ps1"
      Type = "File"
      Contents = "`"Die aktuelle Uhrzeit: `$(Get-Date -Format t)`""
    }
  }
}
```

In dieser Konfiguration wird per *Import-DSCResource* eine DSC-Ressource aus dem Modul *PSDesiredStateConfiguration* importiert, so dass die etwas lästige Warnung nicht mehr erscheint. Beim Aufruf der Konfiguration müssen jetzt ein oder mehrere Namen von Computern angegeben werden. Für jeden Computer wird eine eigene Mof-Datei angelegt.

Der folgende Aufruf legt drei Mof-Dateien an: „Server1.mof“, „Server2.mof“ und „Server3.mof.“ Ob die Computer existieren, spielt zu diesem Zeitpunkt noch keine Rolle.

```
HalloDSC -Computername Server1, Server2, Server3
```

Erst der Aufruf von *Start-DSCConfiguration* überträgt die Konfiguration auf alle Computer, für die eine Mof-Datei vorhanden ist. Sollte dabei eine Authentifizierung erforderlich sein, gibt es dafür bei *Start-DSCConfiguration* den *Credential*-Parameter.

Im letzten Schritt soll für das angelegte Verzeichnis eine Freigabe angelegt werden. Grundsätzlich ist das per DSC kein Problem, allerdings ist eine Ressource für eine Freigabe nicht von Anfang an Bestandteil der PowerShell. Genau wie Module werden auch DSC-Ressourcen über die PowerShell Gallery angeboten und werden per *Install-Module* lokal hinzugefügt. Warum Install-Module? Ganz einfach: DSC-Ressourcen liegen als Module vor. Die zuständige Ressource heißt „xSMBShare“.

Der folgende Befehl lädt die DSC-Ressource xSMBShare als Modul von der PowerShell Gallery:

```
Install-Module -Name xSMBShare -Force
```

Da es sich um ein Modul handelt, wird das Modul unter *C:\Program Files\WindowsPowerShell\Modules* abgelegt.

▶ **Hinweis** Die DSC-Ressource *xSmbShare* stammt aus dem DSC Resource Kit von Microsoft, das wie vieles andere auch auf GitHub als Open Source-Projekt zur Verfügung steht. Sie enthält eine Ressource mit dem Namen „MSFT_xSmShare". Da die Ressource als „Friendly Name" den Namen „xSmbShare" besitzt, kann sie auch über diesen Namen angesprochen werden. Auch das unscheinbare „x" hat einen Grund. Es deutet an, dass die Ressource noch „experimentell" ist, wenngleich sie problemlos anwendet werden kann.

Die Ressource *xSmbShare* liegt damit vor. Wie wird sie angewendet? Sie muss per *Import-DSCResource* im Rahmen der Konfiguration importiert werden. Anschließend kann die Ressource wie jede andere Ressource auch verwendet werden. Die erforderlichen Einstellungen sollten zu diesem Zeitpunkt bereits vertraut sein. Per *Ensure* wird festgelegt, dass sie angelegt werden soll, per *Name* wird der Name der Freigabe angegeben und per *Path* der Verzeichnispfad des freizugebenden Verzeichnisses. Auch hier gibt es weitere Eigenschaften, die im Moment aber keine Rolle spielen.

▶ **Hinweis** Sollte sich die ISE darüber beschweren, dass eine Ressource „xSmbShare" nicht gefunden werden kann, das Phänomen ist bekannt. Es geht in der Regel nach ein paar Mal Probieren von alleine weg.

```
configuration HalloDSC
{
  param([String[]]$Computername)

  Import-DscResource -ModuleName PSDesiredStateConfiguration
  Import-DscResource -ModuleName xSmbShare

  node $Computername
  {

    # Anlegen einer Datei
    File TestDir
    {
      Ensure = "Present"
      DestinationPath = "C:\PoshSkripte"
      Type = "Directory"
    }

    # Anlegen eines Verzeichnisses
    File TestFile
    {
       Ensure = "Present"
       DestinationPath = "C:\PoshSkripte\Test.ps1"
       Type = "File"
       Contents = "`"Die aktuelle Uhrzeit: `$(Get-Date -Format t)`""
    }
     # Geht nicht auf Windows 7 und Windows Server 2008 R2
     xSmbShare TestShare
     {
         Ensure = "Present"
         Name = "TestShare"
         Path = "C:\PoshSkripte"
         Description = "Nur ein Test"
     }
  }

}
```

Mit dem Ausführen der Konfiguration wird die Mof-Datei angelegt. Per *Start-DSCConfiguration* wird sie angewendet. Aktuell kann die Konfiguration aber nur auf dem lokalen Computer ausgeführt werden, da ein potenzieller Zielcomputer gleich mehrere Voraussetzungen erfüllen muss:

- Es muss mindestens Windows Server 2012 oder Windows 8.1 vorhanden sein.
- Es muss die passende PowerShell-Version installiert sein. Wird die Ressource zum Beispiel von einem Computer mit WMF 5.1 übertragen, muss auf allen (!) Nodes diese Version installiert sein.
- Die Ressource xSMBShare muss vorhanden sein.
- Die Ausführung von Skripten muss erlaubt sein, da eine Psm1-Datei ausgeführt wird.
- CIM-Remoting muss aktiviert worden sein.

▶ **Hinweis** Die Ressource *xSmbShare-Ressource* besitzt einen kleinen Nachteil, der sich unter Umständen gar nicht auswirkt: Da für das Anlegen der Freigabe interne Functions aus dem *SmbShare*-Modul verwendet werden, setzt sie mindestens Windows Server 2012 beziehungsweise Windows 8.1 voraus. Wird die Konfiguration daher auf einen Windows 7-PC übertragen, sind Fehlermeldungen die Folge. Solche Abhängigkeiten sieht man einer DSC-Ressource nicht direkt an. Sie schränkt die Verwendbarkeit von DSC in einem bezüglich der Windows-Versionen heterogenen Netzwerk etwas ein. Ein weiterer Umstand, den es zu berücksichtigen gilt, ist, dass nachträglich importierte Ressourcen auf jedem Zielsystem ebenfalls vorhanden sein müssen. Sie werden nicht automatisch übertragen. Das ist insofern logisch, da DSC-Ressourcen auf Modulen basieren und diese ebenfalls auf dem Zielsystem vorhanden sein müssen, wenn sie von einem Skript verwendet werden.

Ein weiterer Aspekt, der beim Übertragen von Konfigurationen auf andere Computer berücksichtigt werden muss, ist der Umstand, dass der *Local Configuration Manager*, der die Ressource auf dem Zielcomputer umsetzt, unter dem lokalen Systemkonto ausführt und daher weniger Berechtigungen besitzt.

8.2 Ein wenig Theorie

In diesem Abschnitt stelle ich einige theoretische Konzepte von DSC vor: Das MOF-Format, die DSC-Spracherweiterungen, Ressourcen und den *Local Configuration Manager*, kurz LCM.

8.2.1 MOF

Mit dem *Managed Object Format* (MOF) hat das PowerShell-Team für DSC auf ein bewährtes Format zurückgegriffen. Es wurde mit WMI zur Definition von Klassen

eingeführt und eignet sich gut als plattformübergreifendes Textformat. MOF ist ein internes Format und nicht zu gedacht direkt editiert zu werden. Durch die Ausführung einer Konfiguration wird diese in eine MOF-Datei kompiliert.

8.2.2 DSC-Spracherweiterungen

Gerade erfahrene PowerShell-Anwender sind beim ersten Schritt mit DSC aufgrund der ungewohnten Schreibweise etwas irritiert. Eine Konfiguration hat so gar nichts mit einem vertrauten Skript zu tun. Vor allem scheint es keine „Liste" aller neuen „DSC-Befehle" zu geben. Keine Sorge, diese kurze Irritationsphase geht im Allgemeinen schnell vorbei. Bei DSC wird zwischen einer Spracherweiterung, die bei der PowerShell ab Version 4.0 eingebaut ist, und Functions und Cmdlets aus dem Modul *PSDesiredStateConfiguration* unterschieden. *Start-DSCConfiguration* ist zum Beispiel ein Cmdlet aus diesem Modul. Die Cmdlets und Functions werden in Kap. 10 vorgestellt.

Die Liste der Spracherweiterungen ist sehr klein, denn es sind nur drei Befehle:

- *Configuration*
- *Node*
- *Import-DSCResource*

Darüber hinaus gibt es ein paar automatische Variablen wie *AllNodes*, die in vielen Konfigurationen verwendet werden. Auch diese werden Sie in den folgenden Kapiteln kennenlernen.

8.2.3 Die Rolle der Ressourcen

Die Ressourcen sind die Funktionsbausteine von DSC. Über eine Ressource wird ein bestimmter „Gegenstand" konfiguriert. Eine Ressource besitzt einen Namen und einen Satz von Eigenschaften. Aktionen gibt es nicht. Eine Aktion wird deklarativ festgelegt, indem einer Eigenschaft ein bestimmter Wert zugewiesen wird. Soll der Gegenstand angelegt oder entfernt werden, wird dies über die Eigenschaft *Ensure*, die jede Ressource besitzt, festgelegt.

Ressourcen basieren auf Modulen. Die Ressourcen-Modulverzeichnisse befinden sich im Programme-Verzeichnis unter *WindowsPowerShell\Modules*. Aus der Sicht des Anwenders ist ein DSC-Modul eine Blackbox, der innere Aufbau spielt keine Rolle. Im nächsten Kapitel wird der Aufbau eines DSC-Moduls vorgestellt. So viel seit bereits verraten: Die Logik, die bei der Anwendung einer Ressource auf den Node durch den LCM ausgeführt wird, besteht aus regulären PowerShell-Befehlen. Im einfachsten Fall besteht ein DSC-Ressourcenmodul aus einer Psm1-Datei.

Die PowerShell verfügt auch in der Version 5.1 nur über mehr als ein Dutzend Ressourcen. Gegenüber der Version 4.0 kamen mit *WindowsFeatureSet*, *WindowsOptionalFeatureSet*,

ServiceSet und *ProcessSet* als „Composite Resources“ und *WaitForAll*, *WaitForAny* und *WaitForSome* für die Synchronisierung zwischen Nodes lediglich ein paar speziellere Ressourcen hinzu. Damit kommt man im Allgemeinen nicht weit. Das PowerShell-Team stellt daher weit über 150 Ressourcen im Rahmen des DSC Resource Kits zur Verfügung, das in regelmäßigen Abständen aktualisiert wird. In der Anfangszeit wurde das Resource Kit in Gestalt einer großen Zip-Datei zum Download angeboten. Inzwischen kann es nicht mehr komplett, sondern nur noch in Gestalt einzelner Ressourcen über die PowerShell Gallery geladen werden. Da DSC-Ressourcen als Module vorliegen, geschieht dies über ein *Install-Module*. Eine Ressource muss als Modul auf jedem Node vorhanden sein, der per DSC konfiguriert werden soll. Sollen daher Konfigurationseinstellungen domänenweit verteilt werden, ist etwas Vorüberlegung in diesem Punkt erforderlich. Ad hoc lässt sich diese Aufgabe im Allgemeinen nicht bewerkstelligen.

▶ **Hinweis** Die wichtigste Frage ist natürlich: Wo sind die DSC-Ressourcen mit ihren Eigenschaften dokumentiert? Wo gibt es Beispiele? Nicht in der PowerShell-Hilfe. Die „eingebauten“ Ressourcen werden im MSDN-Portal dokumentiert. Für alle Ressourcen des DSC Resource Kits findet man eine ausführliche Beschreibung mit Beispielen als Teil des GitHub-Portals (https://github.com/PowerShell/DscResources).

Alle lokal vorhandenen Ressourcen werden über das *Get-DSCResource*-Cmdlet aufgelistet. Alle in der PowerShell Gallery verfügbaren Ressourcen listet ein *Find-Module* auf. Da der Modulname nur in Ausnahmefällen mit dem Namen der Ressource identisch ist, gibt es den *DscResource*-Parameter, über den gezielt nach einer Ressource gesucht wird. Tab. 8.1 stellt alle bei der Version 5.0/5.1 eingebauten Ressourcen zusammen (ein * bedeutet, dass es die Ressource erst seit der Version 5.0 gibt).

▶ **Hinweis** Auch die bei der PowerShell „eingebauten“ Standard-Ressourcen stehen in der PowerShell Gallery als Modul mit dem Namen „PSDscResources“ zur Verfügung. Der Vorteil dieses Moduls ist, dass es die neueste Version dieser Ressourcen enthält, die ebenfalls kontinuierlich verbessert werden.

8.2.4 Der Local Configuration Manager (LCM)

Der *Local Configuration Manager*, kurz LCM, ist das Herzstück von DSC. Er übernimmt das sogenannte „heavy lifting“, indem er für die Anwendung einer Ressource auf dem Node zuständig ist. Er ist „out of the box“ einsatzbereit und muss für speziellere Operationen, etwa die Konfiguration im Push-Modus, konfiguriert werden. Das geschieht entweder über eine DSC-Ressource oder die Function *Set-DSCLocalConfigurationManager* aus dem Modul *PSDesiredStateConfiguration*. Ein *Get-DSCLocalConfigurati-*

Tab. 8.1 DSC-Ressourcen, die bei WMF 5.0 von Anfang an dabei sind

DSC-Ressource	Was wird konfiguriert?
Archive	Zip-Dateien.
Environment	Umgebungsvariablen.
Group	Eine lokale Gruppe.
GroupSet (*)	Mehrere Gruppen als „Composite Resource“.
Log	Ein Eintrag in einem Ereignisprotokoll.
Package	Msi-Paket oder Setup.exe, die über die Ressource ausgeführt wird.
ProcessSet (*)	Mehrere zu startende Prozesse als „Composite Resource“.
Registry	Die Registry mit ihren Schlüsseln, Einträgen und Werten.
Script	Führt einen beliebigen Scriptblock aus.
Service	Systemdienste.
ServiceSet (*)	Eine Gruppe von Systemdiensten als „Composite Resource“.
User	Ein lokaler Benutzer.
WaitForAll (*)	Eine Pseudo-Ressource, die eine Synchronisation mit einer Ressource auf einem anderen Node ermöglicht. Mit dieser Ressource geht es erst dann weiter, wenn alle angegebenen Ressourcen vorhanden sind.
WaitForAny (*)	Eine Pseudo-Ressource, die eine Synchronisation mit einer Ressource auf einem anderen Node ermöglicht. Mit dieser Ressource geht es erst dann weiter, wenn eine der angegebenen Ressourcen vorhanden ist.
WaitForSome (*)	Eine Pseudo-Ressource, die eine Synchronisation mit einer Ressource auf einem anderen Node ermöglicht. Mit dieser Ressource geht es erst dann weiter, wenn eine Ressource auf einer Mindestanzahl an Ressourcen vorhanden ist.
WindowsFeature	Ein Windows-Feature.
WindowsFeatureSet (*)	Eine Gruppe von Windows-Features („Composite Resource“).
WindowsOptionalFeature (*)	Ein optionales Feature.
WindowsOptionalFeatureSet (*)	Ein Gruppe von optionalen Features („Composite Resource“).
WindowsProcess	Ein zu startender Prozess.

onManager gibt die aktuellen Einstellungen des LCM aus. Weitere Details zum LCM verrate ich im nächsten Kapitel.

8.2.5 Pull statt Push

Bei allen bisher gezeigten Beispielen wurde die Konfiguration im Push-Modus zu den angegebenen Computern (Nodes) übertragen. Mit dem Pull-Modus gibt es eine Alternative,

die in großen Server-Landschaften besser geeignet ist. In diesem Modus zieht sich jeder Node die für ihn vorgesehenen Konfigurationsdaten von einem Computer, der dazu als Pull-Server konfiguriert wurde. Dieser Computer muss keine besonderen Voraussetzungen erfüllen und ist ein weiterer Server mit Windows Server ab Version 2008 R2 als Betriebssystem. Damit ein Node seine Konfigurationsdaten in regelmäßigen Intervallen zieht, muss der *Local Configuration Manager* (LCM) auf jedem Node entsprechend konfiguriert werden. Wie das in der Praxis umgesetzt wird, wird im nächsten Kapitel besprochen.

8.3 Verwenden von Konfigurationsdaten

Zum Abschluss dieses Kapitels möchte ich das Einführungsbeispiel, das zu Beginn des Kapitels vorgestellt wurde, etwas erweitern. Der Umstand, dass der Inhalt der anzulegenden Ps1-Datei Teil der Konfigurationsdefinition ist, ist etwas unflexibel. Flexibler wäre es, wenn der Inhalt außerhalb der Definition stehen würde. Diese Möglichkeit besteht allgemein in Gestalt von Konfigurationsdaten, die bei der Ausführung der Konfiguration angegeben werden. Auf diese Weise lassen sich diese Daten für mehrere Konfigurationen verwenden. Die Grundlage für Konfigurationsdaten ist eine Hashtable, die einer Variablen zugewiesen wird.

Beispiel

Die folgende Definition für eine Konfiguration legt ebenfalls wieder ein Verzeichnis und eine Datei an. Der Inhalt der Datei stammt dieses Mal aus den Konfigurationsdaten, die beim Ausführen der Konfiguration per *ConfigurationData*-Parameter übergeben werden.

```
$ConfigData = @{
  AllNodes = @(
    @{
      NodeName = "Server1"
      Ps1Content = "`"Heute ist `$(Get-Date -Format dddd), der `$(Get-
Date -Format dd) te um `$(Get-Date -Format t).`""
    }
  )
}
```

Die Konfigurationsdaten bestehen aus einer Hashtable, die den Eintrag „AllNodes“ besitzt. Dieser steht für ein Array von Hashtables. Jede Hashtable muss einen Schlüssel „NodeName“ besitzen, über den die Einstellung einem bestimmten Node zugeordnet wird. Soll die Einstellung für alle Nodes gelten, kann in vielen Situationen für NodeName auch ein „*“ gesetzt werden. In dem Beispiel wird in den Konfigurationsdaten für den Node „Server1“ die Einstellung *Ps1Content* mit einem Wert belegt.

Beispiel

Das folgende Beispiel zeigt die Definition einer Konfiguration, die die Konfigurationsdaten aus dem letzten Beispiel verwendet.

```
configuration HalloDSC
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $AllNodes.Nodename
  {
    File Dir1
    {
      Ensure = "Present"
      DestinationPath = "C:\TestDir"
      Type = "Directory"
    }

    File File1
    {
      Ensure = "Present"
      DestinationPath = "C:\TestDir\TestFile.ps1"
      Type = "File"
      Contents = $AllNodes.Ps1Content
    }
  }
}

$ConfigData = @{
  AllNodes = @(
    @{
      NodeName = "MobilServer"
      Ps1Content = "`"Heute ist `$(Get-Date -Format dddd), der `$(Get-
Date -Format dd) te um `$(Get-Date -Format t).`""
     }
  )
}
```

Da in den Konfigurationsdaten auch der Name des Node über die Eigenschaft *Nodename* festgelegt wird, ändert sich bei Verwendung von Konfigurationsdaten auch die Art und Weise, wie der Node-Name festgelegt wird.

Der folgende Aufruf führt die Konfiguration aus:

```
HalloDSC -ConfigurationData $ConfigData
```

Der folgende Aufruf überträgt die Konfiguration auf den Zielcomputer, in diesem Fall den lokalen Computer:

```
Start-DSCConfiguration -Path HalloDSC  -Wait -Verbose –Force
```

8.4 Warum DSC?

Zum Abschluss dieser kurzen Übersicht möchte ich, ebenfalls in der gebotenen Kürze, auf ein paar Gründe eingehen, warum das PowerShell-Team sich so etwas wie DSC überhaupt ausgedacht hat. Dafür sind mehrere Gründe verantwortlich:

1. Bei Windows Server gab es bislang keine einheitliche Konfigurationstechnik. Während Unix- und Linux-Server über Textdateien konfiguriert werden, gab es bei Windows in der Vergangenheit einen Mix aus verschiedenen Schnittstellen (WMI, RPC, ADSI, Registry usw.). Mit DSC gibt es erstmals eine einheitliche Plattform für das Verteilen von Konfigurationseinstellungen.
2. Die Welt der Server-Konfiguration entwickelt sich weiter. Das Konzept der Skripte stammt noch aus der Anfangsära und ist unter modernen Gesichtspunkten in vielen Bereich überholt beziehungsweise es besitzt Nachteile, die gerade in Umgebungen, in denen Konfigurationszustände in kurzen Abständen bereitgestellt werden sollen, nicht mehr adäquat sind.
3. Das PowerShell-Team musste zur Kenntnis nehmen, dass die Botschaft des „Wir können alles skripten, wenn wir nur wollen“ trotz zahlreicher Vereinfachungen, die im Laufe der Versionen in die PowerShell eingeführt wurden, nur eine Minderheit an Administratoren erreicht hat.
4. Der Erfolg von Konfigurationsframeworks wie *Chef* und *Puppet* in der Linux-Welt hat deutlich gemacht, dass es für deklarative Konfigurationsframeworks einen Bedarf gibt.

Es versteht sich von selber, dass DSC nur für einen kleinen Teil der Administratoren in Frage kommt. Für viele mag es wie eine Antwort auf eine Frage sein, die niemand gestellt hat. Diese Sichtweise greift aber zu kurz, denn in der Welt der IT sind in den vergangenen Jahren neue Anwendungsbereiche hinzugekommen. Dazu gehört der Bereich der Bereitstellung von Anwendungen, insbesondere von Webanwendungen. Die großen Internetfirmen aktualisieren ihre Anwendungen in sehr kurzen Intervallen und müssen auf Lastschwankungen flexibel durch das Bereitstellen weiterer (virtueller) Server oder Container reagieren. Diese modernen Anforderungen lassen sich durch klassische Skripte nicht mehr optimal abbilden, zumal das erforderliche Know-how oft einfach nicht vorhanden ist. Eine textuelle, deklarative Beschreibung des gewünschten Konfigurationszustands erfordert lediglich domänenspezifisches Wissen und die Fähigkeit, mit einem Editor oder einem anderen Werkzeug umgehen zu können.

8.5 Zusammenfassung

Mit der *Desired State Configuration* (DSC) steht seit der Version 4.0 ein alternativer Ansatz zur Verfügung, über den sich Konfigurationsänderungen auf andere Computer anwenden lassen. Anstatt die gewünschten Konfigurationsänderungen wie in der Vergangenheit durch

eine Folge von Skriptbefehlen festzulegen, wird der gewünschte Zustand mit Hilfe einer eigens für DSC entwickelten Beschreibungssprache festgelegt. Der Vorteil ist, dass dank DSC eine einheitliche Beschreibungssprache zur Verfügung steht, deren Anwendung nicht das typische Scripting-Know-how voraussetzt. Als Zielcomputer, in der DSC-Terminologie „Node“ genannt, kommt grundsätzlich jeder Computer in Frage, auf dem WMF ab Version 4.0 installiert ist. Da die Anwendung einer DSC-Konfiguration auf CIM-Remoting basiert, muss der Node für CIM-Remoting aktiviert worden sein. Dank „DSC for Linux“ kann der Node theoretisch auch ein Linux-Computer sein. Theoretisch deswegen, weil auch entsprechende Ressourcen vorhanden sein müssen. Eine DSC-Konfiguration wird entweder im Push-Modus auf die einzelnen Nodes übertragen oder jeder Node wird so konfiguriert, dass er in regelmäßigen Intervallen Konfigurationen von einem dedizierten „Server“ zieht – im einfachsten Fall genügt dazu eine SMB-Freigabe. DSC spielt auch im Zusammenhang mit Azure eine Rolle, in dem VMs per DSC konfiguriert werden.

DSC in der Praxis

9

Zusammenfassung

In diesem Kapitel geht es ausschließlich um Praxisbeispiele für den Einsatz von DSC. Neben den Kern-Ressourcen, die bei der PowerShell 5.0/5.1 fest eingebaut sind, wird an kleinen Beispielen gezeigt, wie virtuelle Hyper V-Maschinen und ein IIS-Webserver per DSC eingerichtet werden. Die meisten Beispiele sind einfach gehalten, da sie den Einsatzzweck einer einzelnen Ressource veranschaulichen sollen. Einige Beispiele verwenden Konfigurationsdaten. Einige Beispiel verwenden der Einfachheit halber Verzeichnispfade, die noch angepasst werden müssten. Wenn ein fiktiver Remote-Computer angesprochen wird, heißt dieser „Server1“. Wie zu allen Kapiteln sind auch die Beispiele in diesem Kapitel Teil der Beispielsammlung. Die Bezugsquellen werden in der Einleitung genannt.

9.1 Auspacken von Zip-Dateien

Die *Archive*-Ressource ist für das Auspacken von Zip-Dateien in ein Verzeichnis zuständig. Da der LCM lokal ausgeführt wird und der Zugriff auf eine Freigabe einen Benutzername und ein Kennwort voraussetzt, die nicht unverschlüsselt übergeben werden sollten, befindet sich die Zip-Datei im Idealfall bereits auf dem Zielcomputer. Die wichtigsten Eigenschaften der *Archive*-Ressource sind *Path* und *Destination*.

Beispiel

Das folgende Beispiel lädt mit Hilfe der *Script*-Ressource und einfachen PowerShell-Befehlen eine Zip-Datei aus dem Internet und kopiert die Dateien in ein lokales Verzeichnis:

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_9

```
<#
 .Synopsis
 Zip-Datei auspacken per DSC
#>

configuration ArchiveTest
{
  param([String]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $Computername
  {
    # Per Script-Ressource Zip-Datei herunterladen
    Script DownloadZip
    {
      # muss $true zurückgegeben
      TestScript = { !(Test-Path -Path C:\Temp\Ps1Skripte.zip) }
      # Rückgabewert muss Hashtable sein - Result spielt keine Rolle?
      GetScript = {
        @{Result = ""}
      }
      SetScript = {
        $Url = http://www.activetraining.de/Downloads/Ps1Skripte.zip
        Write-Verbose "*** Downloading $Url"
        $LocalPath = "C:\Temp\Ps1Skripte.zip"
        $WC = New-Object -TypeName System.Net.WebClient
        $WC.DownloadFile($Url, $LocalPath)
      }
    }

    Archive Ps1Zip
    {
      Ensure = "Present"
      Path = "C:\Temp\Ps1Skripte.zip"
      Destination = "C:\Ps1Skripte"
      Force = $true
      DependsOn = "[Script]DownloadZip"
    }
  }
}
```

Aufgerufen und angewendet wird die Konfiguration wie folgt:

```
ArchiveTest -ComputerName Server1
Start-DSCConfiguration -Path ArchiveTest -Verbose -Wait -Force
```

9.2 Umgebungsvariablen anlegen

Die *Environment*-Ressource ist für das Anlegen von Umgebungsvariablen zuständig. Geht es darum, die *Path*-Umgebungsvariable zu erweitern, gibt es dafür die Eigenschaft *Path*, die einen *$true*-Wert erhält. Die wichtigsten Eigenschaften der *Environment*-Ressource sind *Name*, *Path* und *Value*. Da der LCM unter dem System-Konto ausführt, wird eine Umgebungsvariable immer auf der Computerebene angelegt.

Beispiel

Das folgende Beispiel legt ein Verzeichnis „C:\PsTools“ an und kopiert eine bereits vorhandene Exe-Datei in dieses Verzeichnis. Anschließend wird die *Pat*h-Umgebungsvariable um den Verzeichnispfad erweitert, so dass die Exe-Datei ohne Voranstellen des Verzeichnispfades aufgerufen werden kann.

```
<#
 .Synopsis
 Umgebungsvariable auf System-Ebene anlegen
#>

configuration EnvTest
{
  param([String]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $Computername
  {
    File Dir1
    {
      Ensure = "Present"
      DestinationPath = "C:\PsTools"
      Type = "Directory"
    }

    File File1
    {
       Ensure = "Present"
       SourcePath = "F:\Tools\Smtp4Dev.exe"
       DestinationPath = "C:\PsTools\Smtp4Dev2.exe"
       Type = "File"
       DependsOn = "[File]Dir1"
    }

    Environment Env1
    {
      Ensure = "Present"
      Name = "Path"
      Path = $true
      Value = "C:\PsTools"
      DependsOn = "[File]Dir1"
    }
  }
}
```

Aufgerufen und angewendet wird die Konfiguration wie folgt:

```
EnvTest -Computername Localhost
Start-DSCConfiguration -Path EnvTest -Wait -Verbose –Force
```

▶ **Hinweis** Damit eine angelegte Umgebungsvariable innerhalb einer PowerShell-Sitzung sichtbar wird, muss diese neu gestartet werden. Es kann sogar erforderlich sein, den Explorer neu zu starten, da die PowerShell-Hostanwendung in der Regel mit dem Explorer gestartet wird und von dem Elternprozess die noch nicht aktualisierte Umgebung übernimmt.

9.3 Dateien und Verzeichnisse anlegen

Die *File*-Ressource ist für das Anlegen von Dateien und Verzeichnissen zuständig. Die wichtigsten Eigenschaften sind *DestinationPath* und *Type*, über die festgelegt wird, ob eine Datei oder ein Verzeichnis angelegt werden soll. Der Inhalt einer Datei wird über die *Contents*-Eigenschaft festgelegt. Soll über die *File*-Ressource eine vorhandene Datei angelegt werden, wird diese über die *SourcePath*-Eigenschaft angegeben. Die Angabe eines UNC-Pfades ist leider nicht so einfach wie es sein könnte, da der direkte Zugriff auf eine Freigabe an den fehlenden Berechtigungen des LCM scheitert.

Beispiel

Das folgende Beispiel legt eine Profilskriptdatei „Profile.ps1“ im Benutzerprofil unter *$env:userprofile/Documents/WindowsPowerShell* an, ihr Inhalt wird über externe Konfigurationsdaten festgelegt.

```
<#
 .Synopsis
 Anlegen einer Profilskriptdatei
#>

configuration ProfileTest
{
  param([String]$Username="Administrator")
  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  node $AllNodes.NodeName
  {
    File ProfileDir
    {
      Ensure = "Present"
      DestinationPath = "C:\Users\$Username\Documents\WindowsPowerShell"
      Type  "Directory"
      Force = $true
    }

    File ProfileFile
    {
      Ensure = "Present"
      DestinationPath =
"C:\Users\$Username\Documents\WindowsPowerShell\Profile.ps1"
      Contents = $AllNodes.ProfileContent
      Type = "File"
      DependsOn = "[File]ProfileDir"
     }
  }
}

$ConfigData = @{
  AllNodes = @(
     @{
       NodeName = "MobilServer"
       ProfileContent = "`$Host.PrivateData.ErrorBackgroundColor =
'White'"
     }
  )
}
```

Aufgerufen und angewendet wird die Konfiguration wie folgt:

```
ProfileTest -Username PsUser -ConfigurationData $ConfigData
Start-DSCConfiguration -Path ProfileTest -Wait -Verbose -Force
```

An diesem kleinen Beispiel wird einer der Vorteile von DSC deutlich. Es spielt keine Rolle, ob die anzulegende Konfigurationseinstellung bereits vorhanden ist oder nicht, da die Abfrage als Teil der Ressource bei deren Umsetzung ausgeführt wird.

9.4 Lokale Benutzer und Gruppen anlegen

Für das Anlegen lokaler Benutzer und Gruppen sind die Ressourcen *User* und *Group* zuständig. Die wichtigsten Eigenschaften beider Ressourcen sind *Username*, *FullName* und *Password*. Für das Passwort muss ein *PSCredential*-Objekt übergeben werden, auch wenn die *Username*-Eigenschaft in diesem Fall keine Rolle spielt. Da ein Kennwort ohne weitere Maßnahmen unverschlüsselt in der Mof-Datei gespeichert wird, erscheint eine Fehlermeldung. Um diese zu vermeiden, muss die Konfigurationsdefinition mit externen Konfigurationsdaten ausgeführt werden, in denen die Eigenschaft *PSDscAllowPlainTextPassword* im *AllNodes*-Knoten auf *$true* gesetzt wird. Dies ist aber nur eine Notlösung. Besser ist es, ein Zertifikat für die Verschlüsselung des Kennworts zu verwenden. Diese Technik wird im Kap. 10 gezeigt.

Beispiel

Das folgende Beispiel legt eine Gruppe „PsUser“ an, zu der zwei ebenfalls zuvor angelegte lokale Benutzerkonten „PsUser1“ und „PsUser2“ hinzugefügt werden.

```
configuration UserGroupAnlegen
{
  param([String[]]$Computername, [PSCredential]$UserPwCred)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $Computername
  {
   User User1
   {
      Ensure = "Absent"
      UserName = "PsUser1"
      FullName = "Peter Monadjemi 1"
      PasswordNeverExpires = $true
      Password = $UserPwCred
   }

   User User2
   {
      Ensure = "Present"
      UserName = "PsUser2"
      FullName = "Peter Monadjemi 2"
      PasswordNeverExpires = $true
      Password = $UserPwCred
   }
```

```
    Group Group1
    {
       Ensure="Absent"
       GroupName="PsUser"
       Members = "PemoUser1", "PemoUser2"
       DependsOn = "[User]User2"
    }
   }
}
```

Da für die Benutzerkonten jeweils ein Kennwort festgelegt werden soll, muss die Konfiguration einen *PSCredential*-Parameter besitzen. Damit die direkte Übergabe eines Kennworts möglich ist, müssen Konfigurationsdaten angelegt werden:

```
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText –Force
$UserPwCred = [PSCredential]::New("Dummy", $PwSec)

$ConfigData = @{
    AllNodes = @(
    @{
      # NodeName darf kein * sein
      NodeName = "Win7B"
      PSDscAllowPlainTextPassword=$true
    }
  )
}
```

Bei der Ausführung der Konfiguration wird das *PSCredential*-Objekt in Gestalt der Variablen *UserPwCred* zusammen mit den Konfigurationsdaten übergeben:

```
UserGroupAnlegen -Computername Server1 -UserPwCred $UserPwCred -
ConfigurationData $ConfigData
```

Wenn die Benutzerkonten auf einem anderen Computer angelegt werden sollen, muss auch beim Ausführen von *Start-DSCConfiguration* ein passendes *PSCredential*-Objekt übergeben werden.

9.5 Registry-Schlüssel anlegen

Registry-Schlüssel und Einträge in der Registry werden mit der *Registry*-Ressource angelegt. Die wichtigsten Eigenschaften der *Registry*-Ressource sind *Key*, *ValueName* und *ValueData*. Ein wichtiger Aspekt ist der Umstand, dass der LCM nicht unter einem Benutzerkonto ausgeführt wird und daher das Anlegen eines Schlüssels unter *HKey_Current_User* ohne Wirkung bleiben würde. Um dies zu vermeiden, muss die Eigenschaft *PsDscRunAsCredential* mit einem *PSCredential*-Objekt belegt werden, dessen *Username*-Eigenschaft den aktuellen Benutzernamen enthält. Da wieder ein Kennwort im Spiel ist, muss mit externen Konfigurationsdaten und der Eigenschaft *PsDscAllowPlainTextPassword* gearbeitet werden. Die Hintergründe werden im Kap. 10 erläutert.

Beispiel

Das folgende Beispiel legt unter *HKey_Current_User* einen Schlüssel mit zwei Einträgen an.

```
<#
 .Synopsis
 Einen Registry-Schlüssel per DSC anlegen
 .Description
 PsDscRunAsCredential ist erforderlich
#>

configuration CreateRegKeys
{
  param([PSCredential]$Credential)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $AllNodes.NodeName
  {
    Registry Key1
    {
      Ensure = "Present"
      Key = "HKey_Current_User\Software\PsKurs"
      ValueName = "StartTermin"
      ValueData = "1.4.2017"
      ValueType = "String"
      PSDscRunAsCredential = $Credential
   }
   Registry Key2
   {
      Ensure = "Present"
      Key = "HKey_Current_User\Software\PsKurs"
      ValueName = "AnzahlTeilnehmer"
      ValueData = "7"
      ValueType = "Dword"
      PsDscRunAsCredential = $Credential
    }
  }
}
```

Aufgrund der erwähnten LCM-Problematik und dem Umstand, dass es explizit erlaubt werden muss, dass ein Kennwort unverschlüsselt in die Mof-Datei geschrieben wird, kommen wieder Konfigurationsdaten ins Spiel:

```
$ConfigData = @{
  AllNodes = @(
     @{
        NodeName = "Localhost"
        PsDscAllowPlainTextPassword = $true
      }
    )
}
```

Die Konfiguration wird wie folgt angewendet:

```
$Username = "Administrator"
$PwSec = "demo+123" | ConvertTo-Securestring -AsPlainText -Force
$Cred = [PSCredential]::new($Username, $PwSec)
CreateRegKeys  -Credential $Cred -ConfigurationData $ConfigData

Start-DSCConfiguration -Path CreateRegKeys -Wait -Verbose -Force
```

9.6 Windows-Feature installieren

Mit der *WindowsFeature*-Ressource werden auf einem Windows Server Features hinzugefügt. Es ist daher eine Ressource, die nur unter Windows Server angewendet werden kann. Da das Hinzufügen eines Features eine überschaubare Angelegenheit ist, ist auch die Ressource einfach gestrickt. Die einzige Eigenschaft der *WindowsFeature*-Ressource ist *Name*, über die der Name des Features festgelegt wird.

Beispiel

Das folgende Beispiel fügt zu einem Node das Feature „PowerShell Web Access“ hinzu.

```
<#
 .Synopsis
 Windows-Feature anlegen per DSC
#>

configuration FeatureTest
{
  param([String[]]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration


  node $Computername
  {
    WindowsFeature Feature1
    {
      Ensure = "Present"
      Name = "WindowsPowerShellWebAccess"
      LogPath = "C:\FeatureInstall.log"
    }
  }
}
```

Ausgeführt wird die Konfiguration wie folgt:

```
FeatureTest -Computername Server1

Start-DSCConfiguration -Path FeatureTest -Wait –Verbose
```

9.7 Prozesse starten

Mit Hilfe der *WindowsProcess*-Ressource wird ein Prozess auf dem Remote-Computer gestartet. Es gelten allerdings auch hier die üblichen Einschränkungen, die in erster Linie darin bestehen, dass der Prozess in keiner interaktiven Session ausgeführt wird und daher zum Beispiel kein Anwendungsfenster anzeigen und Eingaben per Tastatur entgegennehmen kann. Die wichtigsten Eigenschaften der *WindowsProcess*-Ressource sind *Arguments*, *Path* und *StandardOutputPath*.

Beispiel

Das folgende Beispiel startet auf dem Remote-Computer „Cmd.exe“ und legt fest, dass die Ausgabe des Kommandos in eine Textdatei geschrieben wird.

```
<#
 .Synopsis
 Einen Prozess starten per DSC
#>

configuration StartProcessTest
{
  param([String[]]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $Computername
  {
    WindowsProcess Test
    {
       Ensure = "Present"
       Arguments = "/c dir C: /s"
       Path = "C:\Windows\System32\Cmd.exe"
       StandardOutputPath = "C:\CmdOutput.txt"
    }
  }
}
```

Da die Konfiguration auf einen Remote-Computer angewendet wird, der nicht Teil der Domäne ist, werden für *Start-DSCConfiguration* Credentials für diesen Computer benötigt. Ausgeführt wird die Konfiguration wie folgt:

```
StartProcessTest -ComputerName Server1

$Username = "Administrator"
$PwSec = "demo+123" | ConvertTo-Securestring -AsPlainText –Force
$Cred = [PSCredential]::new($Username, $PwSec)

Start-DSCConfiguration -Path StartProcessTest -Wait -Verbose -Credential
$Cred
```

9.8 Systemdienste einrichten

Über die *Service*-Ressource wird ein Systemdienst eingerichtet beziehungsweise werden die Eckdaten eines vorhandenen Dienstes geändert. Die wichtigsten Eigenschaften der *Service-Ressource* sind *Name*, *BuiltInAccount*, *StartupType*, *DisplayName*, *Description* und *Path*.

Beispiel

Das folgende Beispiel richtet einen Systemdienst ein, der als Exe-Datei mit dem Namen „Zitatedienst“ auf dem Node-Computer vorliegt. Der Starttyp wird auf „Manual“ gesetzt, so dass der Dienst nicht automatisch startet. Die Exe-Datei ist Teil der Beispiele für dieses Buch. Der Pfad für die Exe-Datei muss angepasst werden.

```
<#
 .Synopsis
 Einen Systemdienst installieren
#>

configuration SetupQuoteService
{
  param([String]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $Computername
  {
    Service QuoteService
    {
      Ensure = "Present"
      Name = "Zitatedienst"
      BuiltInAccount = "LocalSystem"
      StartupType = "Manual"
      Description = "Originelle StarTrek-Zitate aus TOS"
      DisplayName = "TOS-Zitatedienst"
      Path = "C:\PoshSkripte\DSC\Zitatedienst.exe"
    }
  }
}
```

Die Konfiguration wird wie folgt ausgeführt:

```
SetupQuoteService -Computername Localhost
Start-DSCConfiguration -Path SetupQuoteService -Wait -Verbose –Force
```

9.9 DSC-Log-Meldungen schreiben

Mit der *Log*-Ressource werden während der Umsetzung einer Konfiguration Meldungen in das lokale DSC-Eventlog geschrieben. Der Name des Eventlogs ist „Microsoft-Windows-DSC/Analytic“. Dieses Eventlog muss aber zuerst per *Wevtutil.exe* auf jedem Computer aktiviert werden, ansonsten verpufft das Schreiben wirkungslos. Die einzige Eigenschaft der *Log*-Ressource ist *Message*.

Das Aktivieren des Eventlogs übernimmt der folgende Aufruf von *Wevtutil* mit dem *sl*-Kommando, der in einer Administrator-PowerShell durchgeführt werden muss:

```
Wevtutil sl Microsoft-Windows-DSC/Analytic /e:true
```

Beispiel

Das folgende Beispiel schreibt eine Meldung in das DSC-Eventlog. Es setzt voraus, dass das Eventlog bereits per *Wevtutil* aktiviert wurde.

```
<#
 .Synopsis
 Log-Eintrag schreiben per DSC
#>

configuration LogTest
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node localhost
  {
    Log Log1
    {
     # Erscheint im Microsoft-Windows-DSC/Analytic-Log als Teil von
Message
     Message = "Alles klar mit DSC"
    }
  }
}
```

Die folgenden Befehle führen die Konfiguration aus:

```
LogTest
Start-DSCConfiguration -Path LogTest -Verbose -Wait
```

Ab jetzt werden bei der nächsten DSC-Aktion Meldungen geschrieben, die sich per *Get-WinEvent* ausgeben lassen:

```
Get-WinEvent -LogName Microsoft-Windows-DSC/Analytic -Oldest | Select-
Object Id, TimeCreated, Message
```

Da es sich um ein Analytical-Log handelt, muss der *Oldest*-Parameter gesetzt werden.

► **Tipp** Die Meldungen werden (natürlich) in das Eventlog jenes Computers geschrieben, auf dem die Konfiguration angewendet wird. Um per *Get-WinEvent* Abfragen gegen andere Computer auszuführen, ist es einfacher, dies per *Invoke-Command* zu erledigen, da die Abfrage per *Computername*-Parameter auf klassischem WMI basiert und daher immer dann scheitert, wenn aus irgendeinem Grund auf der Gegenseite die Voraussetzungen nicht erfüllt sind.

Geht es um Meldungen, die in ein beliebiges Eventlog geschrieben werden, wird die Ressource *xWinEventLog* von der PowerShell Gallery benötigt.

9.10 Beliebige Befehle ausführen

Für den Fall, dass keine Ressource passend ist beziehungsweise etwas aktiv im Rahmen der Anwendung einer Konfiguration durchgeführt werden soll, gibt es die *Script*-Ressource. Sie besteht aus drei Pflichteigenschaften: *TestScript*, *GetScript* und *SetScript*. Auch wenn alle drei Eigenschaften vom Typ *String* sind, wird ihnen eine Zeichenkette zugewiesen, die einen oder mehrere PowerShell-Befehle enthalten kann. Per *TestScript* wird festgelegt, ob die Befehle, die unter *SetScript* angegeben sind, überhaupt ausgeführt werden. Die per *TestScript* ausgeführten Befehle müssen *$true* oder *$false* zurückgeben. *$false* führt dazu, dass die per *SetScript* festgelegten Befehle ausgeführt werden. Per *GetScript* wird eine Hashtable zurückgegeben, die einen Schlüssel mit dem Namen „Result“ besitzen muss. Der Wert, der dem Schlüssel zugewiesen wird, spielt keine Rolle, da *GetScript* intern gar nicht ausgeführt wird.

- **Hinweis** Die *Script*-Ressource gilt eher als Notbehelf. Ein kleiner Nachteil ist, dass sich die Befehle nicht debuggen lassen. Sollen per *Script*-Ressource Aktivitäten ausgeführt werden, für die es auch eine spezialisierte Ressource gibt, sollte diese verwendet werden.

- **Hinweis** In der Regel sollen die per *SetScript* ausgeführten Befehle auch irgendwelche Konfigurationsdaten enthalten. Diese werden angesprochen, indem der Node-Variablen ein „using:“ vorangestellt wird.

Beispiel

Das folgende Beispiel verwendet die Script-Ressource, um per DSC eine Datei aus dem Internet zu laden und in ein festgelegtes Verzeichnis zu verschieben. Die URL und der Name der lokalen Datei werden in den Konfigurationsdaten festgelegt. Die URL wird über „$using:Node.Url“ angesprochen:

```
<#
 .Synopsis
 Ein Beispiel für die Script-Ressource
#>

configuration ScriptBeispiel
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $AllNodes.NodeName
  {

  Script DownloadZip
  {
    # muss $false zurueckgegeben
    # $env:temp geht nicht
    TestScript = {
            $LocalFile = $using:Node.LocalFile
            Test-Path -Path C:\Temp\$LocalFile
    }
```

```
    # Rueckgabewert muss Hashtable sein - Result spielt keine Rolle
    GetScript = {
        @{Result = $true}
    }

    SetScript = {
      # Geht nicht
      # $Url = $Node.Url
      $Url = $using:Node.Url
      $LocalPath = "C:\Temp"
      Write-Verbose "*** Downloading $Url nach $LocalPath ***"
      if (!(Test-Path -Path $LocalPath))
      {
          md $LocalPath | Out-Null
      }
      try
      {
        $WC = New-Object -TypeName System.Net.WebClient
        $WC.DownloadFile($Url, "$LocalPath\$($using:Node.Localfile)")
      }
      catch
      {
        Write-Verbose "Fehler beim Download: $_"
      }
    }
  }
  }
}
```

Die Ressource verwendet Konfigurationsdaten:

```
$ConfigData = @{
  AllNodes = @(
    @{
    NodeName = "Server1"
    Url = "http://www.activetraining.de/Downloads/Ps1Skripte.zip"
    LocalFile = "PoshSkripte123.zip"
   }
  )
}
```

Die folgenden Befehle führen die Konfiguration aus:

```
ScriptBeispiel -ConfigurationData $ConfigData
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText -Force
$Server1Cred = [PSCredential]::new("Administrator", $PwSec)
Start-DSCConfiguration -Path ScriptBeispiel -Wait -Verbose -Credential
$Server1Cred -Force
```

9.11 Einen Webserver einrichten

Das Einrichten eines Webservers ist ein Bereich, in dem DSC seine Stärken ausspielen kann. Eine der häufigsten Anforderungen im modernen IT-Alltag von Internet-Firmen dürfte das Bereitstellen einer Webanwendung sein, und sei es nur zu Testzwecken. DSC ist

für diese Art von Anforderung geradezu prädestiniert, da sich die Konfiguration eines Webservers sehr gut textuell beschreiben lässt. Im einfachsten Fall müssen ein „paar" Dateien in ein Verzeichnis kopiert, ein virtuelles Verzeichnis angelegt und Berechtigungen für das Verzeichnis gesetzt werden, so dass der Webserverdienst auf dieses Verzeichnis zugreifen darf. Kommt der IIS zum Einsatz, wird dieser als Feature hinzugefügt. In der Praxis müssen allerdings weitere Details berücksichtigt werden, so dass bereits ein einfaches Beispiel relativ umfangreich wird.

▶ **Hinweis** Auch für das Einrichten eines Apache Webservers kommt natürlich DSC in Frage. Allerdings gibt es (Stand: April 2017) keine fertige Ressource für Windows. Im Verzeichnis *Demos\DSC* des Open Source-PowerShell-Projekts unter GitHub findet sich ein Anschauungsbeispiel für die Installation unter Linux.

Für die IIS-Konfiguration wird das *xWebAdministration*-Modul benötigt, das per *Install-Module* installiert wird. Es umfasst mehrere Ressourcen, die alle Aspekte einer Webserver-Konfiguration abdecken. Das folgende Beispiel setzt das Modul allerdings nicht voraus, da es über die Script-Ressource ein *Install-Module* ausführt. Anschließend werden der IIS und eine Reihe von Subfeatures hinzugefügt, ein AppPool und eine Website angelegt und die Website als WebApp konfiguriert. Angaben wie der Name der Website oder die Pfade der beteiligten Dateien werden in die Konfigurationsdaten ausgelagert. Die zu konfigurierende Webanwendung ist sehr einfach gehalten und besteht lediglich aus einer Html-Datei und einer Bitmap:

```
<#
 .Synopsis
 Eine Web-App einrichten per DSC
#>

# Fügt das xWebAdministration-Modul hinzu
configuration SetupWebAdminModul
{
  param([String[]]$ComputerName)

  Node $ComputerName
  {
    # xWebAdministration auf dem Node hinzufügen
    Script InstallxWebAdminResource
    {
      GetScript = { @{Result = $true} }

      TestScript = {
         (Get-InstalledModule -Name xWebAdministration -ErrorAction
Ignore) -ne $null
      }

      SetScript = {
        Install-Module -Name xWebAdministration -Force
      }
    }
  }
}
```

```
# Richtet die Website mit WebApp ein
configuration SetupWebApp
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xWebAdministration

  node $AllNodes.NodeName
  {
    # Alle Webserver bezogenen Features hinzufügen
    $i = 0
    foreach($Feature in $Node.FeatureListe)
    {
    $i++
    WindowsFeature "WebFeature$i"
    {
      Ensure = "Present"
      Name = $Feature
    }
  }

  # App Pool einrichten
  xWebAppPool PoshAppPool
  {
    Ensure = "Present"
    Name = $Node.WebAppPoolName
    State  = "Started"
  }

  # Website-Verzeichnis anlegen
  File WebsiteDir
  {
    Ensure = "Present"
    DestinationPath = $Node.WebSitePath
    Type = "Directory"
  }

  xWebSite PoshWebSite
  {
    Ensure = "Present"
    Name = $Node.WebsiteName
    BindingInfo = MSFT_xWebBindingInformation
    {
        Protocol = "Http"
        Port = $Node.Port
      }
      PhysicalPath = $Node.WebsitePath
      State = "Started"
      DependsOn = @("[xWebAppPool]PoshAppPool","[File]WebsiteDir")
    }

  xWebApplication PoshWebApp
  {
    Ensure = "Present"
    Name = $Node.WebAppName
    WebSite = $Node.WebsiteName
    WebAppPool = $Node.WebAppPoolName
    PhysicalPath = $Node.WebSitePath
  }
```

```
  File DefaultHtml
  {
    Ensure = "Present"
    Type = "File"
    DestinationPath = $Node.DefaultHtmlPath
    SourcePath = $Node.DefaultHtmlSourcePath
  }

  File DefaultBitmap
  {
      Ensure = "Present"
      Type = "File"
      DestinationPath = $Node.DefaultImagePath
      SourcePath = $Node.DefaultImageSourcePath
  }
 }
}
```

Die Konfigurationsdaten enthalten die variablen Parameter wie die Namen der zu installierenden Webserver-Features, Name und Verzeichnis der Website usw.

```
$ConfigData = @{
  AllNodes = @(
  @{
    FeatureListe = @("Web-Server", "Web-Mgmt-Tools", "Web-Default-Doc",
     "Web-Dir-Browsing", "Web-Http-Errors", "Web-Static-Content",
     "Web-Http-Logging", "Web-Stat-Compression", "Web-Filtering",
     "Web-CGI", "Web-ISAPI-Ext", "Web-ISAPI-Filter")
    NodeName="Server1A"
    WebAppName = "PoshApp"
    WebAppPoolName = "PoshAppPool"
    WebSiteName = "PoshWebSite"
    WebSitePath = "C:\inetpub\wwwroot\poshsite"
    WebApplicationName = "PoshWebApp"
    WebVirtualDirectoryName = "PoshApp"
    Port = 8008
    DefaultHtmlPath = "C:\inetpub\wwwroot\poshsite\default.html"
    DefaultHtmlSourcePath = "C:\inetpub\wwwroot\iisstart.htm"
    DefaultImageSourcePath = "C:\inetpub\wwwroot\iis-85.png"
    DefaultImagePath = "C:\inetpub\wwwroot\poshsite\iis-85.png"
  }
  )
}
```

Nach den umfangreichen Vorbereitungen beschränkt sich die Ausführung der Konfiguration wieder einmal auf einen simplen Aufruf. Die erste Konfiguration soll lediglich das *xWebAdministration*-Modul hinzufügen:

```
SetupWebAdminModul -ComputerName Server1
```

Der nächste Aufruf konfiguriert die Webanwendung:

```
SetupWebApp -ConfigurationData $ConfigData
```

Für *Start-Configuration* werden Administrator-Credentials benötigt:

```
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText –Force
$Server1Cred = [PSCredential]::new("Administrator", $PwSec)

Start-DSCConfiguration -Path SetupWebApp -Credential $Server1ACred -Wait
-Verbose –Force
```

Ging alles gut, sollte die Eingabe von „http://localhost/poshsite“ auf dem Computer eine kleine Html-Seite mit dem IIS-Logo anzeigen.

9.12 Eine Hyper-VM einrichten

Warum sollte man eine VM per DSC und nicht per Skript einrichten? Das HyperV-Modul umfasst (Stand Server 2016) immerhin die stolze Zahl von 178 Cmdlets. Zum Beispiel weil sich die gewünschte Konfiguration leichter erstellen lässt, weil sie auch von Menschen ohne PowerShell-Kenntnisse festgelegt werden kann und weil sich eine DSC-Konfiguration oder gleich die Mof-Datei auch im Rahmen einer kleinen GUI oder maschinell erstellen lässt.

Für das Anlegen einer Hyper-V-VM wird das *xHyper-V*-Modul benötigt, das der *Install-Module* hinzugefügt wird. Es enthält unter anderem die *xVMHyperV*-Ressource, die über ein Dutzend Eigenschaften anbietet. Weitere Ressourcen aus dem Modul sind *xVhd*, *xVMSwitch*, *xVhdFileDirectory* und *xVMDvdDrive* für das Anhängen eines DVD-Laufwerks. Im Projektportal *https://github.com/PowerShell/Hyper-V* werden die einzelnen Ressourcen mit Beispielen ausführlich beschrieben. Wie arbeitet die Ressource intern? Sie verwendet (natürlich) die Cmdlets aus dem Hyper-V-Modul, da dies die naheliegende Variante ist.

Beispiel

Das folgende Beispiel legt eine VM mit ein paar Eckdaten an. Einige der Konfigurationsdaten, aber nicht alle, wie der Pfad der Vhdx-Datei, werden in Konfigurationsdaten ausgelagert. In der Praxis würde man sämtliche Konfigurationsdaten extern festlegen, damit die Konfiguration flexibel eingesetzt werden kann.

```
<#
 .Synopsis
 Hyper VM per DSC anlegen
#>

$ConfigData = @{

  AllNodes = @(
    @{
      NodeName = "PoshServer"
      VhdPath = "E:\HyperV\Virtual Hard Disks\WindowsServer2012R2.vhdx"
      VhdSize = 64GB
    }
  )
}
```

```
configuration SetupVM
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xHyper-V

  node $AllNodes.NodeName
  {
    xVMHyperV DSCServerVm
    {
      Ensure = "Present"
      Name = "DSCServer"
      Generation = 1
      StartupMemory = 1024MB
      EnableGuestService = $true
      ProcessorCount = 2
      SwitchName = "ExternesNetzwerk"
      VhdPath = $Node.VhdPath
      State = "Running"
    }
  }
}
```

Ausgeführt wird die Konfiguration wie folgt:

```
SetupVm -ConfigurationData $ConfigData
Start-DscConfiguration -Path SetupVm -Wait -Verbose –Force
```

9.13 Zusammenfassung

Die Beispiele, die in diesem Kapitel vorgestellt wurden, haben mit den kleinen Beispielen der DSC-Einführung nicht viel gemeinsam. Es stellt sich bei DSC schnell heraus, dass man das traditionelle Scripting mit seinen kleinen und größeren Herausforderungen gegen etwas Neues austauscht, bei dem es ebenfalls um das Lösen kleinerer und größerer Herausforderungen, den Umgang mit Unzulänglichkeiten im Konzept und funktional nicht ausgereiften Bausteinen geht. Trotzdem bietet DSC im Vergleich zum klassischen Scripting-Ansatz eine deutlich flexiblere Herangehensweise. Eine DSC-Konfiguration ähnelt deutlich mehr einer universellen Beschreibungssprache für Konfigurationszustände als es ein Skript jemals sein kann.

DSC für (etwas) Fortgeschrittene

10

Zusammenfassung

In diesem Kapitel stelle ich die etwas fortgeschritteneren Themen im Zusammenhang mit der Desired State Configuration (DSC) vor. Es geht unter anderem um den Umgang mit Konfigurationsdaten, das Speichern von Kennwörtern in einer Mof-Datei, das Einrichten eines Pull Servers und die Konfiguration des LCM. Auch das Erstellen von Ressourcen wird am Ende des Kapitels kurz behandelt.

10.1 Hinzufügen von DSC-Ressourcen

Die PowerShell umfasst nur einen Grundstock von etwa einem Dutzend Ressourcen. Für den Praxiseinsatz ist das natürlich nicht ausreichend. Ressourcen für Exchange Server, Hyper-V, IIS und Windows-Server-Funktionalitäten sind nicht dabei. Fehlende Ressourcen müssen als Module per *Install-Module* nachgeladen werden. Innerhalb einer Konfiguration werden sie per *Import-DSCResource* geladen. Letzteres ist kein Command, sondern ein Befehl, der nur innerhalb einer Konfiguration erlaubt ist.

Mit dem Erscheinen von WMF 4.0 wurden mehrere Dutzend Ressourcen von Microsoft unter dem Namen „DSC Resource Kit“ zusammengefasst und als Zip-Datei zur Verfügung gestellt. Später wurde auch das Resource Kit auf Open Source „umgestellt“, der Quellcode steht auf *GitHub* zur Verfügung: https://github.com/PowerShell/DscResources. Seitdem kann jeder, der sich berufen fühlt, an einzelnen Ressourcen mitarbeiten, Fehler in den Projektforen melden und seine Verbesserungen als Pull Requests zur Verfügung stehen. Alle paar Monate wird eine aktuelle Version des DSC Resource Kits durch Microsoft freigegeben. Um auf dem Laufenden zu bleiben, muss man lediglich den Blog des PowerShell-Teams in seinem RSS-Leseprogramm abonnieren (http://blogs.msdn.com/b/powershell). Mit jedem Update kommen neue Ressourcen hinzu, alleine im März 2017-Update waren es 19 Stück. Damit stehen im Rahmen des Resource Kits inzwischen weit über 400 Ressourcen zur Verfügung.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_10

▶ **Hinweis** Wer sich durch die verschiedenen Links klickt, landet früher oder später im GitHub-Portal des „PSDSCResources"-Projekts. Dieses Projekt umfasst lediglich die bei WMF „eingebauten" Ressourcen, die ebenfalls als Open Source-Projekt entwickelt werden.

Alle DSC-Ressourcen werden über die PowerShell Gallery angeboten. Die Ressourcen aus dem Resource Kit besitzen den Tag „DSCResourceKit". Der Buchstabe „x" gibt an, dass es sich um eine „experimentelle" Ressource handelt. Der Einsatz geschieht auf eigenes Risiko, es gibt für diese Ressource keinen Support durch Microsoft.

Beispiel

Der folgende Befehl gibt die Namen aller aktuell dort zur Verfügung stehenden Module aus.

```
Find-Module -Repository PSGallery -Tag "DSCResourceKit"
```

Beispiel

Der folgende Befehl gibt zu dem Modul auch die Namen der Ressourcen aus, die es umfasst.

```
Find-Module -Repository PSGallery -Tag "DSCResourceKit" | Sort-Object
Name | Select-Object Name, @{n="Ressourcen";e={($_ | Select-Object -
Expand Includes).DSCResource -join ","}}
```

Beispiel

Wer sich für Statistiken interessiert, der folgende Befehl gibt die Module sortiert nach der Anzahl der Ressourcen aus.

```
Find-Module -Repository PSGallery -Tag "DSCResourceKit" | Sort-Object
Name | Select-Object Name, @{n="Count";e={@($_ | Select-Object -Expand
Includes).DscResource.Count}} | Sort-Object -Descending Count
```

Dass bei einigen Modulen die Zahl 0 erscheint, liegt vermutlich daran, dass der Autor die Metadaten nicht „vorschriftsgemäß" ausgefüllt hat.

Bliebe noch zu klären, wie eine Ressource hinzugefügt wird. Über *Install-Module* und den Namen des Moduls. Alle lokal verfügbaren Ressourcen werden per *Get-DSCResource* aufgelistet.

10.2 Ressourcenabhängigkeiten festlegen

Soll zwischen zwei Ressourcen eine Abhängigkeit hergestellt werden, so dass Ressource A voraussetzt, dass Ressource B umgesetzt wurde, gibt es dafür bei jeder Ressource die Eigenschaft *DependsOn*. Sie erhält als Wert den Typ und den Namen der Ressource, von der diese Ressource abhängig ist. Mehrere Ressourcen werden per Komma getrennt.

Beispiel

Im folgenden Beispiel setzt die *xWebSite*-Ressource voraus, dass die *WindowsFeature*-Ressource angewendet wurde, über die der IIS-Webserver in einer Minimalkonfiguration hinzugefügt wird. Über eine dritte Ressource wird eine Html-Datei im Webserververzeichnis angelegt.

```
configuration WebsiteSetup
{

  Import-DscResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xWebAdministration

  node Localhost
  {

    File DefaultPage
    {
      Ensure = "Present"
      DestinationPath = "C:\Webserver\htdocs\Default.htm"
      Contents = "<H3>Alles klar mit DSC!</H3>"
    }

    xWebsite PoshSite
    {
      Ensure = "Present"
      Name = "PoshSite"
      State = "Started"
      PhysicalPath = "C:\Webserver\htdocs"
      BindingInfo = MSFT_xWebBindingInformation
                    {
                     Protocol = "HTTP"
                     Port = 8000
                    }
      DependsOn = "[WindowsFeature]IIS"
    }

    WindowsFeature IIS
    {
      Ensure = "Present"
      Name = "Web-Server"
      IncludeAllSubFeature = $true
    }
   }
}
```

10.3 Den LCM konfigurieren

Der *Local Configuration Manager* (LCM) ist der Motor von DSC. Er ist Teil des WMF und legt die Arbeitsweise eines Nodes fest. Er sorgt für die Umsetzung von Konfigurationen, die in Gestalt von Mof-Dateien im Pull- oder Push-Modus auf den Node übertragen werden. Der LCM bietet eine Fülle von Einstellungen, die über die *Get-DSCLocalConfigurationManager*-Function geliefert werden.

Das Ändern einer Einstellung geschieht nicht direkt, sondern indirekt per DSC. Dazu muss eine Konfiguration mit dem Attribut *DSCLocalConfigurationManager* ausgestattet

werden. Über die spezielle Ressource *Settings*, auf die kein Name folgt, werden die Einstellungen festgelegt. Das Ausführen der Konfiguration führt dazu, dass eine Mof-Datei mit dem Namen „Localhost.meta.mof“ angelegt wird. Die in dieser Datei enthaltenen Änderungen werden per *Set-DscLocalConfigurationManager*-Function auf den Node angewendet. Über den *Path*-Parameter wird der Pfad des Verzeichnisses angegeben, in dem sich die Meta.Mof-Datei befindet.

▶ **Hinweis** *Get-DSCLocalConfigurationManager* liefert ein klassisches *CimInstance*-Objekt, dessen Eigenschaften entsprechend über die *CimInstanceProperties*-Eigenschaft einzeln abgefragt werden können.

Beispiel

Das folgende Beispiel setzt die Einstellung „RebootIfNeeded“ des LCM auf *true*.

```
<#
 .Synopsis
 LCM für Reboot konfigurieren
#>

[DSCLocalConfigurationManager()]
configuration LCMSetup
{
  Node Localhost
  {
    Settings
    {
      RebootNodeIfNeeded = $true
      ConfigurationMode = "ApplyOnly"
      ActionAfterReboot =  "ContinueConfiguration"
    }
  }
}
```

Die Konfiguration wird angewendet:

```
LCMSetup
Set-DscLocalConfigurationManager -Path .\LCMSetup -Verbose
```

Und wieder abgefragt:

```
Get-DscLocalConfigurationManager
```

10.4 Umgang mit Konfigurationsdaten

Verwenden mehrere Konfigurationen dieselben Daten, bietet es sich an, diese in den Konfigurationsdaten auszulagern. Ein einzelner Satz von Konfigurationsdaten basiert auf einer Hashtable-Variablen, die beim Ausführen der Konfiguration dem Parameter

ConfiguationName übergeben wird. Die Hashtable muss eine Eigenschaft „AllNodes“ besitzen. Der Wert ist ein Array mit weiteren Hashtable-Elementen. Jede dieser Hashtable besitzt in der Regel einen Eintrag „NodeName“, über den der Name des Nodes festgelegt wird, auf den sich die Einträge der Hashtable beziehen sollen. Die Namen der weiteren Einträge sind beliebig. Sollen alle Knoten angesprochen werden, wird für *NodeName* ein „*“ eingetragen. Innerhalb der Konfiguration werden die Konfigurationsdaten über die Variable *AllNodes* und innerhalb eines Nodes über die Variable *Node* angesprochen.

10.4.1 Konfigurationsdaten unterschiedliche Nodes zuordnen

Enthalten die Konfigurationsdaten Daten, die nur für einen bestimmten Node bestimmt sind, werden diese in einer Hashtable abgelegt, über deren *NodeName*-Eigenschaft der Node ausgewählt wird. Auch wenn innerhalb der Konfiguration ein Node über die Variable *Node* und der Name des Nodes über die Variable *NodeName* angesprochen werden, erfolgt die Zuordnung automatisch, so dass jeder Node die für ihn bestimmten Daten aus den Konfigurationsdaten erhält.

Beispiel

Das folgende Beispiel zeigt, wie sich per DSC Verzeichnisse anlegen lassen, deren Pfade über Konfigurationsdaten festgelegt werden. Jedem der beiden Nodes „Server1“ und „Server2“ wird ein eigener Verzeichnispfad zugeordnet. In jedem der beiden Verzeichnisse soll eine Datei angelegt werden. Ihr Name wird daher in den Konfigurationsdaten in dem Bereich abgelegt, der für alle Nodes verwendet wird.

Die Konfigurationsdaten sind wie folgt aufgebaut:

```
$ConfigData = @{
  AllNodes = @(
  @{
     NodeName = "*"
     DateiName = "DSCTest.txt"
   }
  @{
    NodeName = "Server1"
    VerzPfad = "C:\Server1Test"
   }
  @{
    NodeName = "Server2"
    VerzPfad = "C:\Server2Test"
  }
  )
}
```

Innerhalb der Konfiguration müssen die Konfigurationsdaten angesprochen werden. Das geschieht über die Variable *AllNodes*, die immer einen Wert besitzt, wenn der Konfiguration Konfigurationsdaten zugewiesen werden. Über ein „$AllNodes.NodeName“ werden die Namen aller verwendeten Nodes einzeln angesprochen, eine künstliche Wiederholung per *ForEach* ist daher nicht erforderlich.

```
<#
 .Synopsis
 Konfigurationsdaten - Beispiel 1'
#>

configuration ConfigTest1
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $AllNodes.NodeName
  {
    Log ConfigTest
    {
      Message = "Erstelle Verzeichnis {0} auf Node: {1}" -f
$Node.VerzPfad, $NodeName
    }

    File TestDir
    {
      Ensure  = "Present"
      DestinationPath = $Node.VerzPfad
      Type = "Directory"
    }

    File TestFile
    {
      Ensure  = "Present"
      DestinationPath = Join-Path -Path $Node.VerzPfad -ChildPath
$Node.DateiName
      Type = "File"
      Contents = "Alles mit DSC!"
    }
  }
}
```

10.4.2 Einzelne Nodes auswählen

Der Umstand, dass über die Hashtable einem Node beliebige Eigenschaften zugeordnet werden können, macht das Auswählen und damit das Anwenden einer Ressource innerhalb einer Konfiguration für bestimmte Nodes sehr einfach. Dazu muss lediglich an die *AllNodes*-Variable die Erweiterungsmethode *Where* angehängt werden, auf die ein beliebiger Filterausdruck folgt. Innerhalb des Filters wird das Node-Objekt über $_ angesprochen. Der folgende Befehl wählt nur Nodes aus, deren *Rolle*-Eigenschaft den Wert „Spezial" besitzt:

```
Node ($AllNodes.Where{$_.Rolle -eq "Spezial"}).NodeName
```

Beispiel

Im folgenden Beispiel soll die Ressource *File* für das Anlegen einer Datei nur auf bestimmte Nodes angewendet werden. Die Nodes erhalten dazu über die Konfigurationsdaten unter anderem eine Eigenschaft „Rolle". Die Konfigurationsdaten sind wie folgt aufgebaut:

```
$ConfigData = @{
  AllNodes = @(
  @{
    NodeName = "Localhost"
    NodeNr = 1
    Rolle = "Spezial"
    Pfad = "C:\LocalhostTest"
  }
  @{
    NodeName = "Server1"
    NodeNr = 2
    Rolle = "Spezial"
    Pfad = "C:\Server1Test"
  }
  @{
    NodeName = "Server2"
    NodeNr  = 3
    Rolle = "Allgemein"
  }
  @{
    NodeName = "Server3"
    NodeNr  = 4
    Rolle = "Spezial"
    Pfad = "C:\Server3Test"
  }
 )
}
```

Innerhalb der Konfigurationsdefinition wird über *AllNodes* und *Where* eine Auswahl der Nodes anhand ihrer *Rolle*-Eigenschaft durchgeführt:

```
configuration ConfigTest
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node ($AllNodes.Where{$_.Rolle -eq "Spezial"}).NodeName
  {
    File TestDir
    {
      Ensure  = "Present"
      DestinationPath = $Node.Pfad
      Type = "Directory"
    }
  }
}
```

Das Kombinieren von Eigenschaften, die für alle Nodes gelten, mit Eigenschaften, die für bestimmte Nodes gelten, ist mit den Konfigurationsdaten sehr einfach. Es müssen keine Abfragen durchgeführt werden, DSC nimmt die Zuordnung automatisch vor.

10.4.3 Allgemeine Eigenschaften in den Konfigurationsdaten festlegen

Die Konfigurationsdaten können auch allgemeine Daten enthalten, die keinem Node zugeordnet werden. Dazu muss lediglich einem beliebigen Namen ein beliebiger Wert zugeordnet

werden, der auch ein Array oder eine weitere Hashtable sein kann. Der auf diese Weise festgelegte Wert wird innerhalb der Konfiguration über die Variable *ConfigurationData* abgefragt.

Beispiel

Das folgende Beispiel zeigt eine Konfiguration, über die eine Datei auf mehreren Nodes angelegt wird. Der Inhalt der Datei stammt aus den Konfigurationsdaten. Die Konfigurationsdaten sind wie folgt aufgebaut:

```
$ConfigData = @{

  DateiInhalt = "Alles klar mit DSC!"

  AllNodes = @(
   @{
    NodeName = "*"
    Dateipfad= "C:\DSCTest.txt"
    }
   @{
    NodeName = "Server1"
  }
   @{
    NodeName = "Server2"
   }
  )
}
```

Die Konfiguration ist wie folgt aufgebaut:

```
configuration ConfigTest3
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $AllNodes.NodeName
  {
   File TestFile
   {
     Ensure  = "Present"
     DestinationPath = $Node.Dateipfad
     Type = "File"
     Contents = $ConfigurationData.DateiInhalt
   }
  }
}
```

10.4.4 Konfigurationsdaten, die nur allgemeine Einstellungen enthalten

Auch ohne Node-Namen können Konfigurationsdaten verwendet werden, in diesem Fall wird lediglich ein leerer *AllNodes*-Eintrag eingefügt. Der für das Anlegen der Mof-Datei erforderliche Nodename wird in diesem Fall entweder „fest verdrahtet“ oder über den Parameter *Computername* festgelegt.

Beispiel

Im folgenden Beispiel wird lediglich der Inhalt einer anzulegenden Datei in den Konfigurationsdaten abgelegt. Die Konfiguration wird damit sehr übersichtlich:

```
$ConfigData = @{
  AllNodes = @()
  DateiInhalt = "Alles klar mit DSC!"
}
```

Innerhalb der Konfiguration wird „DateiInhalt“ über die Variable *ConfigurationData* angesprochen:

```
configuration ConfigTest4
{
  param([String[]]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $Computername
  {
    File TestFile
    {
      Ensure  = "Present"
      DestinationPath = $Node.Dateipfad
      Type = "File"
      Contents = $ConfigurationData.DateiInhalt
    }
  }
}
```

10.4.5 Konfigurationsdaten mit strukturierten Werten

Dass die allgemeinen Konfigurationsdaten beliebig strukturiert sein und damit zum Beispiel auch ein Array mit Hashtables enthalten können, macht das letzte Beispiel in diesem Abschnitt deutlich. In den Konfigurationsdaten werden die Details für zwei anzulegende Dateien abgelegt.

```
$ConfigData = @{

  Dateien  = @(
    @{
      Nr = 1
      Pfad = "C:\DSCTest1"
      Inhalt = "Alles klar mit DSC!"
    }

    @{
      Nr = 2
      Pfad = "C:\DSCTest2"
      Inhalt = "Mit DSC geht alles klar!"
     }
    )

    AllNodes = @()

}
```

Da der Nodename nicht in den Konfigurationsdaten festgelegt wird, wird dieser über den Parameter *ComputerName* festgelegt.

Die Konfiguration ist wie folgt aufgebaut:

```
configuration ConfigTest5
{
  param([String[]]$Computername)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  Node $Computername
  {
    foreach($Datei in $ConfigurationData.Dateien)
    {
      File "Datei$($Datei.Nr)"
      {
        Ensure  = "Present"
        DestinationPath = $Datei.Pfad
        Type = "File"
        Contents = $Datei.Inhalt
      }
    }
  }
}
```

Konfigurationsdaten sind bei DSC eine praktische Angelegenheit, sie lassen sich flexibel und mit wenig Mehraufwand einsetzen. Der Name eines Nodes sollte generell über Konfigurationsdaten und nicht über einen *ComputerName*-Parameter festgelegt werden, da dies die flexiblere Methode ist.

10.5 Kennwörter in einer Konfiguration verwenden

Erfordert eine Ressource ein Kennwort, wird dieses immer als Teil eines *PSCredential*-Objekts übergeben. Dies ist in einer DSC-Konfiguration nicht anders als in einem PowerShell-Skript. Ressourcen, die mit Kennwörtern arbeiten, besitzen daher in der Regel eine Eigenschaft vom Typ *PSCredential*. Anders als Skripte werden Konfigurationen aber nicht ausgeführt, sondern in eine Mof-Datei übersetzt. Das Problem: In der Mof-Datei würde das Kennwort ohne weitere Maßnahmen im Klartext eingetragen. Um dies zu verhindern, wird offiziell ein Zertifikat benötigt, das für die Dokumentsignierung geeignet ist. Der Zusatz „offiziell“ deutet es bereits an: Wie immer gibt es auch ein „Schlupfloch“, das einem diesen kleinen Mehraufwand erspart. Doch der Reihe nach.

Wird einer Eigenschaft einer Ressource ein *PSCredential*-Objekt zugewiesen, das für einen Parameter übergeben wurde, gibt es beim Ausführen der Konfiguration zunächst eine Fehlermeldung vom Typ *System.InvalidOperation* und den Hinweis „Das Konvertieren und Speichern eines verschlüsselten Kennworts als Klartext wird nicht empfohlen“. Weitere Hinweise gibt es online. Die Konfiguration lässt sich in dieser Form nicht umsetzen. Wie sich das Problem lösen lässt, wird in der Fehlermeldung leider nicht verraten.

Es gibt zwei Alternativen, die beide Konfigurationsdaten erfordern. Im einfachsten Fall wird in den Konfigurationsdaten über die Eigenschaft *PSDscAllowPlainTextPassword* festgelegt, dass das Kennwort unverschlüsselt in die Mof-Datei eingetragen werden darf. Die zweite Alternative erfordert etwas mehr Aufwand: In den Konfigurationsdaten wird über die Eigenschaft *CertificateFile* der Pfad einer Zertifikatdatei auf dem Node-Computer angegeben, mit der das Kennwort verschlüsselt wurde. Diese Datei muss auf jedem Node vorhanden sein. Außerdem muss der LCM des Nodes den Thumbprint des Zertifikats kennen. Die Konfigurationsdaten erhalten daher beim ersten Ausführen auch eine Eigenschaft Thumbprint, mit deren Wert der LCM in einem separaten Schritt per *Set-DSCLocalConfigurationManager*-Cmdlet aktualisiert wird.

Im Folgenden werden beide Varianten vorgestellt.

Ausgangspunkt für die folgenden Beispiele ist die Ressource *xADUser* aus dem *xActiveDirectory*-Modul für das Anlegen eines Benutzerkontos im Active Directory. Damit das Benutzerkonto aktiviert werden kann, wird ein Kennwort als Secure String benötigt. Dieses muss in Gestalt eines *PSCredential*-Objekts der Eigenschaft *Password* zugewiesen werden.

Beispiel

Das folgende Beispiel zeigt eine Konfiguration für das Anlegen eines aktiven Benutzerkontos. Die Ausführung der Konfiguration wird zunächst mit einer Fehlermeldung scheitern, da ein *PSCredential*-Objekt involviert ist.

```
<#
 .Synopsis
 Umgang mit Credentials in einer Konfiguration - Beispiel Nr. 1
#>

configuration PasswordConfig1
{
  param(
      [PSCredential]$Credential
 )

  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xActiveDirectory

  node Localhost
  {
    xADUser UserNeu
    {
      Ensure = "Present"
      UserName = "DSCTest1"
      DomainName = "pshub.local"
      DisplayName = "DSC-Test User Nr. 1"
      Enabled = $true
      Password = $Credential
    }
  }
}
```

Der Aufruf führt trotz *PSCredential*-Parameter zu einer Fehlermeldung:

```
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText –Force
$Cred = [PSCredential]::new("Administrator", $PwSec)
PasswordConfig1 -Credential $Cred
```

Der Hintergrund für die Fehlermeldung ist, dass es den Anwendern nicht zu einfach gemacht werden soll, gegen elementare Sicherheitsregeln zu verstoßen. Wer der Meinung ist, dass dies zu vertreten ist, muss dazu lediglich die Einstellung *PSDscAllowPlainTextPassword* in den Konfigurationsdaten der Konfiguration auf *$true* setzen.

Beispiel

Das folgende Beispiel entspricht dem letzten Beispiel, nur dass dieses Mal die Konfiguration mit Hilfe der Konfigurationsdaten umgesetzt wird. Wie ein Blick in die Mof-Datei verrät, wurde das Kennwort, das im *PSCredential*-Objekt verschlüsselt enthalten ist, unverschlüsselt in die Datei geschrieben.

```
<#
 .Synopsis
 Umgang mit Credentials in einer Konfiguration - Beispiel Nr. 2
#>

configuration PasswordConfig2
{
  param([PSCredential]$Credential)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xActiveDirectory

  node Localhost
  {

  xADUser UserNeu
  {
    Ensure = "Present"
    UserName = "DSCTest1"
    DomainName = "pshub.local"
    DisplayName = "DSC-Test User Nr. 1"
    Enabled = $true
    Password = $Credential
  }
 }
}

$ConfigData = @{
    AllNodes = @(
     @{
        NodeName = "Localhost"
        PSDscAllowPlainTextPassword = $true
      }
    )
}
```

Der folgende Aufruft läuft jetzt glatt durch:

```
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText –Force
$Cred = [PSCredential]::new("Administrator", $PwSec)
PasswordConfig2 -ConfigurationData $ConfigData -Credential $Cred
```

Die empfohlene Vorgehensweise besteht darin, das Kennwort mit einem Zertifikat zu verschlüsseln. Dazu muss auf jedem (!) Node ein Zertifikat vorhanden sein, das für die Dokumentverschlüsselung geeignet ist. Ein reguläres SSL-Zertifikat, das eventuell bereits vorhanden ist, genügt daher nicht. Das Zertifikat muss sich in der Ablage „Eigene Zertifikate“ befinden.

▶ **Tipp** Ob ein vorhandenes Zertifikat für die Dokumentverschlüsselung geeignet ist, erfährt man aus den Eigenschaften des Zertifikats, die man sich zum Beispiel im Rahmen der Zertifikatverwaltung anzeigen lassen kann. Eine Alternative ist das *Get-Item*-Cmdlet. Im zurückgegebenen Objekt enthält die *EnhancedKeyUsageList*-Eigenschaft die Angaben über die Einsatzmöglichkeiten des Zertifikats.

Auf dem Node, auf dem die Konfiguration angewendet wird, müssen zwei Voraussetzungen erfüllt sein: Die Cer-Datei mit dem exportierten Public Key muss vorhanden sein. Der LCM muss einmalig mit dem Thumbprint des Zertifikats konfiguriert werden.

Der erste Schritt besteht darin, ein Zertifikat anzulegen, das für die Dokumentverschlüsselung geeignet ist. Unter Windows 10 AE und Windows Server 2016 geht dies sehr einfach über das erweiterte *New-SelfSignedCertificate*-Cmdlet. Für alle anderen Windows-Versionen empfehle ich das PowerShell-Skript „New-SelfSignedCertificateEx.ps1“, das im Rahmen der PowerShell Gallery zur Verfügung steht. Es wird im Kap. 15, in dem es allgemein um das Thema Sicherheit geht, ausführlicher vorgestellt.

Beispiel

Der folgende Aufruf der Function *New-SelfSignedCertificateEx* aus dem Skript „New-SelfSignedCertificateEx.ps1“ legt ein Zertifikat an, das für die Dokumentverschlüsselung geeignet ist. Die Verwendung des Schlüssels wird über die Parameter *KeyUsage* und *EnhancedKeyUsage* festgelegt. Für den ersten Parameter bietet die Function eine Auswahlliste mit Standardverwendungszwecken an. Bei dem zweiten Parameter wird der Wert entweder über eine Bezeichnung in der Landessprache oder als Ziffernfolge (OID-Wert) angegeben. Über den Parameter *StoreLocation* wird das Zertifikat in die Ablage „Computer\Eigene Zertifikate“ verfrachtet. Der CN-Wert spielt keine Rolle.

```
New-SelfSignedCertificateEx -Subject "CN=PsKurs" -StoreLocation
LocalMachine\My -KeyUsage KeyEncipherment -EnhancedKeyUsage
"Dokumentverschluesselung"
```

Im nächsten Schritt muss das Zertifikat mit seinem öffentlichen Schlüssel in eine Cer-Datei exportiert werden, damit es auf die Nodes übertragen werden kann. Ab Windows Server 2012 und Windows 8.1 gibt es dafür das *Export-Certificate*-Cmdlet aus dem *PKI*-Modul. Unter älteren Versionen wird ein Export am einfachsten über die Zertifikatkonsole („Certmgr.msc“ für die benutzerspezifischen Zertifikate) erledigt. Ob das Zertifikat binär oder Base64-codiert exportiert wird, spielt keine Rolle.

Im nächsten Schritt wird die Cer-Datei auf einen Node kopiert und dort zum Beispiel im Verzeichnis „C:\PublicKeys“ abgelegt.

Damit sind alle Voraussetzungen getroffen, damit ein Kennwort als Teil eines *PSCredential*-Objekts verschlüsselt in die Mof-Datei geschrieben werden kann.

Beispiel

Das folgende Beispiel entspricht dem letzten Beispiel, nur dass dieses Mal über die Konfigurationsdaten der öffentliche Schlüssel eines Zertifikats ausgewählt wird, mit dem das Kennwort verschlüsselt und auf dem Node wieder entschlüsselt wird.

```
<#
 .Synopsis
 Umgang mit Credentials in einer Konfiguration - Beispiel Nr. 3
#>

configuration PasswordConfig3
{
  param([PSCredential]$Credential)

  Import-DSCResource -ModuleName PSDesiredStateConfiguration
  Import-DSCResource -ModuleName xActiveDirectory

  node Localhost
  {

    xADUser UserNeu
    {
      Ensure = "Present"
      UserName = "DSCTest2"
      DomainName = "pshub.local"
      DisplayName = "DSC-Test User Nr. 2"
      Enabled = $true
      Password = $Credential
    }

    LocalConfigurationManager
    {
        CertificateID = $Node.ThumbPrint
    }
  }
}

$CertThumb = (dir Cert:\LocalMachine\My -Eku "1.3.6.1.4.1.311.80.1"|
Where Subject -eq "CN=PMServer").Thumbprint

$ConfigData = @{

  AllNodes = @(
      @{
        NodeName = "Localhost"
        CertificateFile = "C:\PublicKeys\PMServer.cer"
        Thumbprint = $CertThumb
      }
  )
}
```

Da die Konfigurationsdaten auch Daten für den LCM enthalten, wird zusätzlich eine Datei mit der Erweiterung „meta.mof“ angelegt. Diese muss auf jedem Node mit dem *Set-DSCLocalConfigurationManager*-Cmdlet umgesetzt werden:

```
Set-DSCLocalConfigurationManager –Path .\PasswordConfig3 -Verbose
```

Jetzt kann die Konfiguration per *Start-DSCConfiguration* ausgeführt werden:

```
Start-DSCConfiguration –Path .\PasswordConfig3 –Verbose –Wait
```

10.6 Einrichten eines Pull Servers

Der Standardbetriebssystemmodus des LCM ist der Push-Modus. In diesem Modus werden Konfigurationsänderungen in Gestalt von Mof-Dateien per *Start-DSCConfiguration*-Cmdlet an einen oder mehrere Clients übertragen. Geht es darum, Konfigurationsänderungen auf eine größere Anzahl an Computern zu übertragen, ist der Pull-Modus etwas praktischer. In diesem Modus „zieht“ sich der LCM in festgelegten Intervallen Konfigurationen von einem festgelegten „Pull Server“. Ein Pull Server ist ein beliebiger Windows-Computer im Netzwerk, der lediglich als Pull Server eingerichtet und im LCM eines Nodes eingetragen wurde.

Es gibt aktuell (bezogen auf die Version 5.1) zwei Varianten für einen Pull Server: Auf der Basis des IIS-Webservers und einer speziellen Erweiterung und auf der Basis einer simplen SMB-Freigabe. Beide Varianten besitzen ihre Vor- und Nachteile. Das Einrichten eines IIS-Pull Servers ist etwas aufwändiger und kann an „Kleinigkeiten“ wie einem ungültigen Zertifikat scheitern. Dafür lassen sich unter anderem Berechtigungen einstellen. Ein SMB-Pull Server setzt lediglich eine Freigabe voraus, bietet aber keine Funktionalitäten wie einen „Reportserver“, an den jede Konfigurationsänderung durch den LCM gemeldet wird, so dass sich Konfigurationsreports abfragen lassen.

In den folgenden Abschnitten wird die erste Variante vorgestellt, da sie in der Praxis die bevorzugte Variante sein dürfte.

▶ **Tipp** Das Einrichten eines Pull Servers ist in der MSDN-Dokumentation ausführlich beschrieben: https://msdn.microsoft.com/en-us/powershell/dsc/pullServer.

10.6.1 Überblick über das Einrichten eines Pull Servers

Die Inbetriebnahme eines Pull Servers besteht allgemein aus drei Schritten:

1. Einrichten des Pull Servers auf einem Server im Netzwerk.
2. Bereitstellen von Modulen und einzelnen Ressourcen in den dafür vorgesehenen Verzeichnissen auf dem Pull Server, so dass diese von den Nodes gezogen werden können.
3. Umstellen des LCM jedes einzelnen Nodes auf den Pull-Modus.

10.6.2 Einrichten eines webbasierten Pull Servers

Voraussetzung für einen webbasierten Pull Server ist der IIS Webserver. Für das komplette Einrichten eines solchen Pull Servers gibt es eine DSC-Ressource mit dem Namen *xDSCWebservice*, die ein Bestandteil des Moduls *xPSDesiredStateConfiguration* ist (nicht zu verwechseln mit dem DSC-Modul PSDesiredStateConfiguration). Dies erspart einiges an Arbeit im Vergleich zum Einrichten eines IIS per DSC und allen erforderlichen Bausteinen.

Schritt 1: Server-Zertifikat anlegen
Dieser Schritt ist nur erforderlich, wenn noch kein Zertifikat für das Einrichten der Https-Verbindung vorliegt. Das Zertifikat kann auf verschiedene Weisen erstellt werden, zum Beispiel direkt in der IIS-Konsole, über das *New-SelfSignedCertificate*-Cmdlet oder über das sehr gute Skript *New-SelfSignedCertificateEx.ps1*, das an verschiedenen Stellen in diesem Buch verwendet wird.

Der folgende Befehl legt ein Zertifikat per *New-SelfSignedCertificate* an:

```
New-SelfSignedCertificate -DnsName PMServer -CertStoreLocation
cert:\LocalMachine\My
```

Das Zertifikat muss in der IIS-Konsole für die Pull Server-Website in die Bindings eingetragen werden.

Sie benötigen den Fingerabdruck des Zertifikats für das Set-up-Skript. Der folgende Befehl gibt diesen aus:

```
dir path cert:\LocalMachine\My | Where-Object Subject -eq "CN=PMServer" |
Select -ExpandProperty ThumbPrint
```

Schritt 2: Registrierungsschlüssel anlegen
Für die Authentifizierung der Nodes beim Pull Server wird eine Zahl benötigt, der Registrierungsschlüssel. Am einfachsten ist es, dafür eine GUID anzulegen:

```
[Guid]::newGuid().Guid
```

bzw.

```
(New-Guid).Guid
```

Die GUID wird für das Set-up-Skript benötigt, das im nächsten Abschnitt vorgestellt wird. Das Skript trägt die Zeichenfolge in die Datei „Registrationkeys.txt“ ein, die im Verzeichnis *$env:ProgramFiles\WindowsPowerShell\DscService* angelegt wird.

Schritt 3: Pull Server per Skript einrichten
Damit sind alle Voraussetzungen erfüllt, um das Set-up-Skript für das Einrichten des Pull Servers ausführen zu können. Das folgende PowerShell-Skript ist zwangsläufig etwas umfangreicher, denn es richtet einen Pull Server auf dem lokalen Computer von A bis Z ein, inklusive der IIS-Installation als Feature. Es setzt voraus, dass ein Server-Zertifikat existiert, dessen Fingerabdruck in das Skript eingetragen wurde. Es enthält die Zeichenfolge, die in die Datei „Registrationkeys.txt“ eingetragen wird.

Als Port wird (willkürlich) 8080 festgelegt. Heißt der Pull Server zum Beispiel „PMServer“ und der Endpunkt „DSCPullServer.svc“, lautet die URL

```
https://pmserver:8080/DSCPullServer.svc
```

Geben Sie diese Adresse in das Adressfeld des Browsers ein, werden die über diesen Endpunkt verfügbaren Ressourcen ausgegeben. Dies ist ein einfacher Test, um festzustellen, ob der Endpunkt und damit der Pull Server für einen Node überhaupt erreichbar ist. Da der IIS im Zusammenhang mit einem Pull Server in der Regel mit einem selbst ausgestellten Zertifikat konfiguriert wird, muss zuvor eine Ausnahmeregel im Browser angelegt werden, damit der Aufruf durchgeführt wird.

▶ **Hinweis** Auch wenn es (natürlich) möglich ist, einen Pull Server mit HTTP zu konfigurieren, sollte diese Option nicht in Betracht gezogen werden. Auch wenn die in diesem Kapitel vorgestellte Variante mit einem selbst ausgestellten Zertifikat ebenfalls keine befriedigende Lösung ist, eine HTTPS-Verbindung ist eine sinnvolle Vorgabe, die man nicht aus Bequemlichkeit aushebeln sollte. Das Umstellen auf HTTP ist aber für die Fehlersuche sehr hilfreich, wenn eine HTTPS-Verbindung an einem Zertifikatfehler scheitern sollte.

Das Skript für das Einrichten eines Https-Pull Servers ist wie folgt aufgebaut:

```
<#
 .Synopsis
 Pull-Server per DSC konfigurieren
#>

configuration SetupDSCPullserver
{
  param(
    [Parameter(Mandatory=$true)]
    [String]$CertThumbPrint,
    [ValidateNotNullOrEmpty()]
    [String]$RegistrationKey
  )

  Import-DSCResource -ModuleName xPSDesiredStateConfiguration
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node Localhost
  {
    WindowsFeature DSCServiceFeature
    {
      Ensure = "Present"
      Name = "DSC-Service"
    }

    xDSCWebService PSDSCPullServer
    {
      Ensure = "Present"
      DependsOn = "[WindowsFeature]DSCServiceFeature"
      EndpointName = "PSDSCPullServer"
      Port = 8088
      PhysicalPath = "$env:SystemDrive\inetpub\PSDSCPullServer"
      CertificateThumbPrint = $CertThumbPrint
      ModulePath =
"$env:ProgramFiles\WindowsPowerShell\DscService\Modules"
      ConfigurationPath =
"$env:ProgramFiles\WindowsPowerShell\DscService\Configuration"
      State = "Started"
      UseSecurityBestPractices = $false
    }

    File RegistrationKeyFile
    {
      Ensure = "Present"
      Type = "File"
      DestinationPath =
"$env:ProgramFiles\WindowsPowerShell\DscService\Registrationkeys.txt"
      Contents = $RegistrationKey
    }
  }
}
```

Beim Aufruf der Konfiguration werden der Fingerabdruck des Zertifikats und der Registrierungsschlüssel übergeben:

```
SetupDSCPullServer -CertThumpPrint $CertThumb -Registrationkey
$RegistrationKey
```

Die Konfiguration wird umgesetzt:

```
Start-DSCConfiguration -Path .\SetupDSCPullServer -Wait -Verbose
```

Je nach Umfang der zu installierenden Komponenten dauert die Ausführung unterschiedlich lang. Dank der Verbose-Meldungen lässt sich alles im Detail nachvollziehen. Lief alles ohne Fehlermeldungen durch, ist der Pull Server eingerichtet. Ein einfacher Test besteht darin, die Https-Adresse im Browser einzugeben. Die Rückgabe besteht aus XML-Text, der die Namen aller Ressourcen umfasst, die der Endpunkt zur Verfügung stellt. Je nach Browser wird dieser Text unterschiedlich formatiert angezeigt (Abb. 10.1).

Ein Test in der PowerShell-Konsole per *Wget* ist ebenfalls sinnvoll:

```
Wget https://pmserver:8080/DSCPullServer.svc
```

Sollte bei diesem Aufruf ein Fehler aufgrund eines ungültigen Zertifikats auftreten, liefert ein „$Error[0].Exception.InnerException“ wertvolle Hinweise zur Fehlerursache. In diesem Fall müssen Sie sich noch einmal die Pull Server-Konfiguration anschauen und sicherstellen, dass der Webservice im IIS mit dem richtigen Zertifikat ausgestattet wurde. Sollte sich die Fehlermeldung schwieriger gestalten, stellt ein vorübergehendes Umschalten auf Http sicher, dass der Pull Server grundsätzlich funktioniert. Für einen Regelbetrieb wird Http grundsätzlich nicht empfohlen.

Im Mittelpunkt des Skripts steht die Ressource xDSCWebservice. Die verwendeten Eigenschaften werden der Übersichtlichkeit halber noch einmal in Tab. 10.1 zusammengestellt.

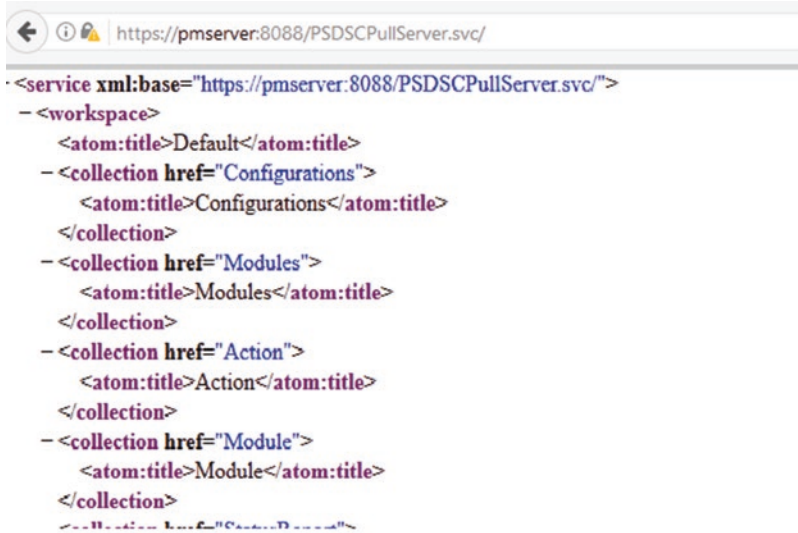

Abb. 10.1 Der Browser-Test ergibt, dass der Pull Server auf Anfragen die richtige Antwort liefert

Tab. 10.1 Die Pull Server-Einstellungen im Überblick

Eigenschaft	Was wird festgelegt?
EndpointName	Der Name des Endpunkts, der mit der Erweiterung .svc an die URL gehängt wird.
Port	Die Portnummer des Dienstes, in Beispielen wird für die HTTPS-Bindung meistens Port 8080 verwendet.
CertificateThumbPrint	Der Fingerabdruck des Server-Zertifikats.
ModulePath	Der Pfad des Modulverzeichnisses, das die Ressourcen enthält, die der Pull Server bereitstellen soll.
ConfigurationPath	Der Pfad für die Mof-Dateien, die der Pull Server bereitstellen soll.
State	Hier wird „Started“ eingetragen, da der Dienst im Allgemeinen sofort laufen soll.
UseSecurityBestPractices	Diese Eigenschaft bestimmt lediglich, ob der Wert „SecureTLSProtocols“ der Eigenschaft *DisableSecurityBestPractices* angewendet wird oder nicht. Besitzt die Eigenschaft den Wert *$true*, darf die Eigenschaft *CertificateThumbPrint* nicht den Wert „AllowUnencryptedTraffic“ besitzen.
DisableSecurityBestPractices	Diese optionale Einstellung kann aktuell nur den Wert „SecureTLSProtocols“ besitzen. Dadurch werden in der Registry unter *HKLM:\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Protocols* die sicheren Protokolle aktiviert und die unsicheren Protokolle deaktiviert.

▶ **Hinweis** Wer nach einer offiziellen Dokumentation der einzelnen Eigenschaften sucht, findet diese (natürlich) im Projektportal der jeweiligen Ressource. Für die Ressource *xDSCWebservice* wäre es https://github.com/PowerShell/xPSDesiredStateConfiguration. Wer sich für die möglichen Werte für eine Eigenschaft wie *DisableSecurityBestPractices* interessiert, findet diese unter anderem im Quellcode. Hier stellt sich heraus, dass für die erwähnte Eigenschaft in der aktuellen Version nur der Wert „SecureTLSProtocols“ erlaubt ist.

Schritt 4: Einrichten des Pull-Verzeichnisses

Die Module und Mof-Dateien, die der Pull Server seinen Nodes zur Verfügung stellen soll, müssen in den Verzeichnissen abgelegt werden, die beim Einrichten des Pull Servers festgelegt wurden. Im Pull-Verzeichnis werden Module als Zip-Dateien und Mof-Dateien direkt als Dateien abgelegt.

Für die Namensgebung gilt die Regel <Modulname>_<Version>.zip, also zum Beispiel „PsKurs_1.0.zip“. Da pro Modul nur eine Version vorhanden sein darf, darf das Modul nicht, wie es bei regulären Modulen üblich ist, einen Ordner mit dem Namen der Versionsnummer enthalten. Die Ordnerhierarchie ist daher ganz einfach, indem die Zip-Datei auf der obersten Ebene die Moduldateien enthält.

Für die Mof-Dateien gilt ebenfalls eine Namensvorgabe. Damit ein Node weiß, welche Mof-Dateien für ihn bestimmt sind und welche nicht, gibt es seit WMF 5.0 das Konzept der Konfigurationsnamen. Ein Konfigurationsname ist ein beliebiger Name, der bei der Konfiguration des LCM für jeden Node eingetragen wird. Im Beispiel in diesem Abschnitt wurde als Konfigurationsname „PoshConfig“ verwendet. Indem eine Mof-Datei den Namen einer Konfiguration erhält, zum Beispiel „Poshconfig.mof“, wird festgelegt, von welchen Nodes sie gezogen wird. Das bedeutet im Umkehrschluss, dass in dieser Konstellation die gesamte Konfiguration eines Nodes in einer Mof-Datei enthalten sein muss. Eine Lösung bieten die partiellen Konfigurationen, die in diesem Buch aus Platzgründen aber nicht behandelt werden. Sie werden in der MSDN-Dokumentation ausführlich beschrieben.

Das war aber noch nicht alles. Sowohl für die Zip- als auch die Mof-Dateien muss eine Prüfsummendatei angelegt werden, die die Erweiterung „checksum“ tragen muss. Sie enthält eine Prüfsumme als einzigen Inhalt. Für das Erstellen der Prüfsumme enthält das *PSDesiredStateConfiguration*-Modul die Function *New-DscChecksum*. Sie wird am einfachsten mit dem Verzeichnispfad als Argument ausgeführt und legt für alle Dateien in dem Verzeichnis eine Checksum-Datei an.

▶ **Tipp** Wer sich die ganze Handarbeit sparen möchte, das DSC-Team bei Microsoft bietet ein Skript an, das alle Schritte für das Veröffentlichen von Modulen und Mof-Dateien auf einem Pull Server zusammenfasst. Es wird im Rahmen der MSDN-Dokumentation, die das Einrichten eines Pull Servers beschreibt, verwendet. Sie finden es unter der folgenden Adresse: https://github.com/PowerShell/xPSDesiredStateConfiguration/blob/dev/DSCPullServerSetup/PublishModulesAndMofsToPullServer.psm1.

10.6.3 Umstellen des LCM auf den Pull-Modus

Damit ein Node seine Konfigurationsdaten von einem Pull Server ziehen kann, muss der LCM konfiguriert werden. Auch das wird per DSC erledigt. Am Anfang sollte man sich auf die erforderlichen Angaben beschränken und alle „Extras“ erst dann hinzufügen, wenn die Minimalkonfiguration funktioniert.

Das folgende Skript richtet auf dem lokalen Computer einen Pull Server ein:

```
<#
 .Synopsis
 LCM für den Pull Server konfigurieren
#>

[DSCLocalConfigurationManager()]
configuration LCMSetup
{
  node localhost
  {
    Settings
    {
      RefreshMode = "Pull"
      # Ist die Default-Einstellung
      ConfigurationMode = "ApplyAndMonitor"
      AllowModuleOverwrite = $true
    }

    ConfigurationRepositoryWeb MeinPullServer
    {
      ServerURL = "https://PMServer:8088/PSDSCPullServer.svc"
      RegistrationKey = "a017fb5b-1808-48f3-acc4-17e6a72138c1"
      ConfigurationNames = @("PoshConfig")
    }
  }
}
```

Ausgeführt wird die Konfiguration wie folgt:

```
LCMSetup
Set-DscLocalConfigurationManager -Path .\LCMSetup -Verbose -Force
```

10.6.4 Einen Pull Server testen

Zum Schluss muss der Pull Server auch getestet werden. Liegt im Configuration-Verzeichnis eine Mof-Datei, deren Name die Namenskonvention erfüllt, wird sie spätestens nach 15 Minuten von jedem konfigurierten Node gezogen und durch den LCM angewendet. Benötigt die Konfiguration Ressourcen, die nicht auf dem Node vorhanden sind, werden diese automatisch geladen, wenn sie sich als Zip-Datei im Modules-Verzeichnis befinden. In beiden Fällen entscheidet der Inhalt der Checksum-Datei darüber, ob die Konfiguration überhaupt gezogen wird.

Beispiel

Das folgende Beispiel zeigt eine absichtlich sehr einfach gehaltene Konfiguration, die bei der Ausführung eine Mof-Datei „Localhost.mof“ erzeugt.

Der folgende Befehl kopiert die Mof-Datei in das dafür vorgesehene Verzeichnis:

```
Copy .\Pulltest\Localhost.mof "C:\Program
Files\WindowsPowershell\DscService\Configuration\Poshconfig.mof"
```

„PoshConfig“ ist der Name, mit dem der Node bereits konfiguriert wurde.

Der nächste Befehl legt die Checksum-Datei an:

```
New-DscChecksum –Path "C:\Program
Files\WindowsPowershell\DscService\Configuration"
```

Jetzt heißt es warten, bis der LCM die Mof-Datei irgendwann zieht.

Wer ungeduldig ist oder im Rahmen von Tests nicht jedes Mal eine Viertelstunde warten kann, führt auf dem Node das *Update-DscConfiguration*-Cmdlet aus, das alle auf einem Pull Server wartenden Konfigurationen zieht und anwendet.

Der folgende Befehl aktualisiert die für einen Node vorliegenden Konfigurationen:

```
Update-DscConfiguration –Wait –Verbose
```

Am Anfang sind die Parameter Verbose und Wait sehr hilfreich, da sich so im Detail nachvollziehen lässt, ob eine Ressource angewendet wurde. Eine weitere zuverlässige Quelle ist das DSC-Eventlog „Operational“, aus dessen Einträgen ebenfalls hervorgeht, ob eine Ressource angewendet wurde.

Über *Get-DSCConfigurationStatus* wird der aktuelle Status eines Nodes ausgegeben. Die Eigenschaft *Status* gibt an, ob das letzte Aktualisieren erfolgreich war oder nicht.

▶ **Tipp** Um *Update-DscConfiguration* dazu zu bewegen, die letzte Konfiguration erneut zu ziehen, muss die Checksum-Datei im Configuration-Verzeichnis gelöscht und neu angelegt werden. Bei *New-DscChecksum* sollte außerdem der *Force*-Parameter gesetzt werden, der bewirkt, dass eine vorhandene Datei überschrieben wird.

10.6.5 Die Rolle der Reportserver

In vielen Beispielen der offiziellen MSDN-Dokumentation und bei anderen Gelegenheiten, wenn es um das Einrichten eines Pull Servers geht, taucht der Begriff „Reportserver“ auf (bei WMF 4.0 war noch von einem „Compliance Server“ die Rede). Dahinter steckt lediglich der Umstand, dass der Pull Server jede von einem Node angeforderte Konfiguration in einer Datenbank speichert, deren Inhalt gezielt per *Invoke-WebRequest*-Cmdlet abgefragt werden kann. Die Rückgabewerte liegen im JSON-Format vor.

10.7 DSC-Diagnose

DSC unterhält mehrere Ereignisprotokolle, in die alle Aktivitäten und vor allem Fehler eingetragen werden. Das Standardprotokoll ist Microsoft-Windows-DSC/Operational. Speziellere Fehler werden in das Analytic-Protokoll geschrieben (Microsoft-Windows-

DSC/Analytic), das zuerst aktiviert werden muss. Die Abfragen werden per Get-WinEvent-Cmdlet durchgeführt.

▶ **Tipp** In einem bereits etwas älteren Blog des PowerShell-Teams wird die Fehlersuche in den DSC-Logs ausführlich beschrieben: https://blogs.msdn.microsoft.com/powershell/2014/01/03/using-event-logs-to-diagnose-errors-in-desired-state-configuration.

Der folgende Befehl gibt den aktuellen Inhalt des Operational-Logs aus:

```
Get-WinEvent -LogName Microsoft-Windows-DSC/Operational
```

Der folgende Befehl gibt den aktuellen Inhalt des Analytic-Logs aus:

```
Get-WinEvent -LogName Microsoft-Windows-DSC/Analytic -Oldest
```

Der *Oldest*-Parameter ist erforderlich, da ein Analytic-Protokoll in der umgekehrten Reihenfolge gelesen werden muss. In der Regel wird die Abfrage aber nicht funktionieren, da das Analytic-Log zuerst aktiviert werden muss.

Bevor sich jemand im Internet auf die Suche macht und Beispiele für das Kommandozeilentool *Wevtutil.exe* findet, speziell für DSC gibt es eine sehr viel einfachere Lösung vom DSC-Team. Es ist das Modul *xDscDiagnostics*, das zuerst per *Install-Module* installiert werden muss. Es umfasst knapp ein halbes Dutzend Functions. Drei Functions sind wichtig:

Get-xDscOperation,
Trace-xDscOperation
und
Update-xDscEventLogStatus.

Per *Get-xDscOperation* werden zunächst die zuletzt durchgeführten Operationen aufgelistet. Zu jeder Operation gibt es eine „Sequence Id“, zum Beispiel die 1. Mit dieser Zahl wird *Trace-xDscOperation* aufgerufen, um alle Meldungen zu dieser Operation zu erhalten. Per *Update-xDscEventLogStatus* kann ein Eventlog aktiviert werden.

Der folgende Aufruf aktiviert das Analytic-Log bei DSC:

```
Update-xDscEventLogStatus -Channel Analytic -Status Enabled
```

Ist das Log bereits aktiviert, schlägt der Aufruf fehl, da das Log für eine Statusänderung zuerst deaktiviert werden muss.

Der aktuelle Status eines Eventlogs kann allgemein per *Wevtutil.exe* abgefragt werden:

```
wevtutil gl Microsoft-Windows-DSC/Analytic
```

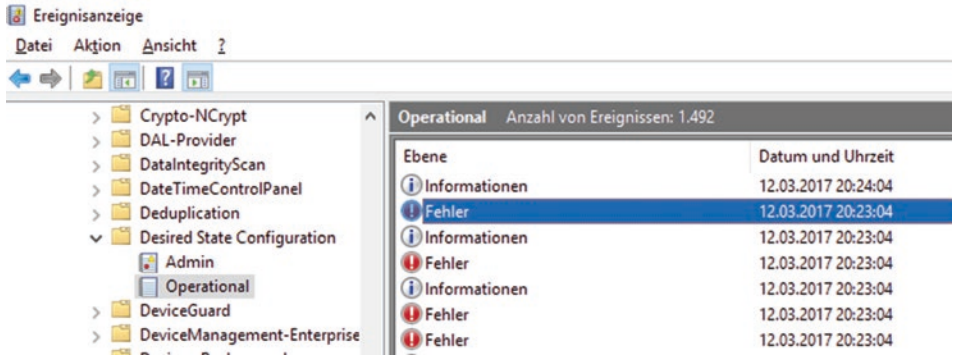

Abb. 10.2 Das DSC-Log in der Ereignisanzeige

Wie geht man vor, um einen DSC-Fehler zu analysieren? Der erste Schritt ist das Ausführen von *Get-xDscOperation*. Im nächsten Schritt werden alle Einträge, deren *Result*-Wert „Failure“ ist, per *Trace-xDscOperation* und der entsprechenden Sequence Id untersucht:

```
Trace-xDscOperation -SequenceID 1
```

Da nur die Einträge mit einem EventType="Error" interessant sind, werden diese gezielt abgefragt, um den Wert der *Message*-Eigenschaft zu erhalten:

```
Trace-xDscOperation -SequenceID 4 | Where-Object EventType -eq "Error" |
Format-List Message
```

Damit erhalten Sie alle DSC-Fehlermeldungen sehr ausführlich und haben nun die Aufgabe, aus den vielen Details die Lösung herauszulesen.

Natürlich wird niemand dazu gezwungen, die Eventlogs in der Kommandozeile zu durchsuchen. Eine Alternative ist natürlich die Ereignisanzeige, die bekanntlich per *Show-EventLog* sichtbar gemacht wird (Abb. 10.2).

10.8 Eigene Ressourcen definieren

Wer möchte (oder muss) und wider Erwarten in der PowerShell Gallery nichts Passendes findet, kann sich mit vertretbarem Aufwand eigene Ressourcen definieren, entweder in Gestalt eines klassischen Modus oder, seit der Version 5.0, auf der Basis einer Klasse. Letztere Variante ist zwar deutlich einfacher in der Umsetzung, setzt aber voraus, dass alle Nodes mit WMF 5.0 oder höher arbeiten. Da es aber generell empfehlenswert ist, bei DSC stets mit der aktuellen Version zu arbeiten, ist dies in der Praxis keine echte Beschränkung.

Damit eine Klasse als DSC-Ressource erkannt wird, muss sie lediglich drei Methoden implementieren: *Set*, *Get* und *Test*. Am wichtigsten sind *Set* und *Test*. Die *Test*-Methode entscheidet über ihren $true/$false-Rückgabewert, ob die *Set*-Methode ausgeführt wird, in der die Konfigurationsänderung durchgeführt wird. Die *Get*-Methode ist nur der Form halber dabei und wird (aktuell) nicht ausgeführt. Die Klasse ist in einer Psm1-Datei enthalten, die von einer Psd1-Datei begleitet wird. Beide werden in einer Zip-Datei zusammengefasst, in deren Name die Versionsnummer enthalten ist, zum Beispiel „ProfileResource_1.0.zip“. Am Ende wird die Zip-Datei im Verzeichnis *C:\Program Files\WindowsPowerShell\DSCService\Modules* abgelegt. Mehr ist im einfachsten Fall nicht zu tun, um eine Ressource über einen Pull Server bereitzustellen.

An dieser Stelle wäre sicherlich ein Beispiel hilfreich. Doch für was könnte man eine Ressource anlegen, die es noch nicht in den Weiten des Internets gibt? Das ist in der Tat etwas knifflig, da alle Standardthemen bereits vergeben sind. Auch sollte eine Ressource sich darauf beschränken, einen Konfigurationszustand herzustellen und sollte keine Schrittfolgen abarbeiten, für die ein Skript besser geeignet wäre. Eine Ressource, die einen Dienst einrichtet, in dem sie die Dienstdatei zuvor aus dem Internet lädt, sollte besser durch einen Skript ersetzt werden, da das Herstellen des Konfigurationszustandes von „zu vielen“ Parametern abhängig ist. Auch wenn sich die Schrittfolge per DSC umsetzen ließe, passen solche Aufgabenstellungen nicht zu DSC.

Im Folgenden wird eine Ressource umgesetzt, die auf einem Node ein Profilskript einrichtet. Über Eigenschaften kann festgelegt werden, ob das Profilskript für alle Benutzer, für alle Hosts oder nur für den aktuellen Benutzer beziehungsweise den aktuellen Host angelegt werden soll. Auch der Inhalt des Skripts kann festgelegt werden. Das Beispiel macht deutlich, dass selbst für eine solch einfache Anforderung bereits einiges an Programmierung erforderlich ist.

Wie wird eine reguläre Klasse zu einer Ressourcendefinition? Über verschiedene „Zusätze“, sprich Attribute. Das Attribut *DSCResource()*sorgt dafür, dass die Klasse als Ressource erkannt wird. Das Attribut *DSCProperty()*macht aus einer Variablen eine Eigenschaft der Ressource.

Das folgende Listing zeigt die Klasse *PSProfileResource*, die sich in der Datei „ProfileResource.psm1“ befindet. Aus Platzgründen wird nur der Rahmen abgebildet, das vollständige Beispiel ist ebenfalls ein Teil der Buchbeispieldateien. Der allgemeine Aufbau einer Ressourcen-Klasse ist einfach; was etwas kniffliger ist, ist die Implementierung, da viele Fallunterscheidungen getroffen werden müssen und sowohl im Set- als auch im Test-Teil zwischen „Ensure=Present“ und „Ensure=Absent“ unterschieden wird, denn per DSC können Konfigurationseinstellungen nicht nur gesetzt, sondern auch wieder entfernt werden.

```
<#
 .Synopsis
 DSC-Ressource für das Anlegen eines Profilskriptes
#>

enum ProfileType
{
  CurrentUserAllHosts
  CurrentUserCurrentHost
  AllUsersAllHosts
  AllUsersCurrentHost
}

enum Ensure
{
  Absent
  Present
}

[DSCResource()]
class xPSProfileResource
{
  [DSCProperty(Key)]
  [ProfileType]$ProfileType

  [DSCProperty(Mandatory)]
  [String]$Username

  [DSCProperty()]
  [String]$Hostname

  [DSCProperty()]
  [String]$ErrorBackgroundColor

  [DSCProperty()]
  [Ensure]$Ensure

  [bool]Test()
  {
   $Result = $false
   switch($this.ProfileType)
   {
     "CurrentUserAllHosts" { }
     "CurrentUserCurrentHost" { }
     "AllUsersAllHosts" { }
      "AllUsersCurrentHost" { }
    }
    return $Result
  }

  [void]Set()
  {
      switch($this.ProfileType)
      {
        "CurrentUserAllHosts" { }
        "CurrentUserCurrentHost" { }
        "AllUsersAllHosts" {
        "AllUsersCurrentHost" { }
       }
     }
    }
  }
```

```
  [xPSProfileResource]Get()
  {
    if ($this.Ensure -eq "Present")  {
        switch($this.ProfileType)
        {
          "CurrentUserAllHosts" {  }
          "CurrentUserCurrentHost" {  }
          "AllUsersAllHosts" { }
          "AllUsersAllHosts" { }
        }
      }
      return $this
    }
}
```

Für die Psm1-Datei wird eine Manifestdatei „ProfileResource.psd1“ benötigt. Sie ist wie folgt aufgebaut:

```
@{
    RootModule = 'ProfileResource.psm1'
    DscResourcesToExport = 'xPSProfileResource'
    ModuleVersion = '1.0'
    GUID = '81624038-5e71-40f8-8905-b1a87afe22d7'
    Author = 'P. Monadjemi'
    CompanyName = 'ActiveTraining'
    Description = 'Legt PowerShell-Profileskripte an'
    PowerShellVersion = '5.0'
}
```

Die folgende Befehlsfolge macht aus der Psm1- und der Psd1-Datei eine Zip-Datei, die in dem für Modul-Ressourcen vorgesehenen Verzeichnis abgelegt wird, und legt auch die für den Pull-Betrieb erforderliche Checksum-Datei an:

```
$CustomResourcePfad = "E:\PoshSkripte\DSC_EigeneRessource"
$DSCModulPfad = "C:\Program Files\WindowsPowerShell\DscService\Modules"
$PSModulPfad = "C:\Program Files\WindowsPowerShell\Modules"

$ModulVersion = "1.0"

$ModulName = [IO.Path]::GetFileNameWithoutExtension((dir -Path
$CustomResourcePfad -Filter *.psd1).Name)

$TmpPfad = Join-Path -Path $env:temp -ChildPath PSCustomResource

if (Test-Path -Path $TmpPfad)
{
  rd $TmpPfad -Recurse –Force
}

# Temporäres Verzeichnis anlegen
md $TmpPfad | Out-Null

# Modul-Dateien in das Modules-Verzeichnis kopieren

$PSModulResourcePfad = Join-Path -Path $PSModulPfad -ChildPath
"$ModulName\$ModulVersion"
```

```
Copy $CustomResourcePfad\*.psm1, $CustomResourcePfad\*.psd1
$PSModulResourcePfad –Force

# Modul-Dateien nach Temp kopieren

Copy $CustomResourcePfad\*.psm1, $CustomResourcePfad\*.psd1 $TmpPfad

# Zip-Datei erstellen
$DestinationPath = "{0}\{1}_{2}.zip" -f $DSCModulPfad, $ModulName,
$ModulVersion

# Wichtig: Die Zip-Datei enthaelt keine Hierarchie, sondern die
Moduldateien direkt
Compress-Archive -Path $TmpPfad\* -DestinationPath $DestinationPath –
Force

# Checksum-Datei erstellen
New-DscChecksum -Path $DSCModulPfad -Force -Verbose
```

In der Praxis ist ein Pester-Test beinahe obligatorisch. Er instanziiert die Ressourcen-Klasse mit der *New*-Methode ihres Typobjekts und ruft die Methoden *Test* und *Set* in unterschiedlichen Konstellationen auf.

Damit liegt die Ressource als Modul vor und kann von jedem Node gezogen werden.

10.8.1 Zusammengesetzte Ressourcen (Composite Resources)

Sobald eine Konfiguration umfangreicher wird, liegt es nahe, mehrere Ressourcen zu einer neuen Ressource zusammenzufassen, um nicht immer wieder alle Ressourcen einzeln definieren zu müssen. Die wichtigsten Einstellungen, die sich von Fall zu Fall ändern, werden der zusammengesetzten Ressource durch Parameter übergeben. Diese „Composite Resources“ gibt es bei WMF von Anfang an. Mit der Version 5.0 wurde die *DependsOn*-Eigenschaft nachgereicht, so dass sich auch Abhängigkeiten zwischen einzelnen Ressourcen abbilden lassen.

Wann werden zusammengesetzte Ressourcen benötigt? Immer dann, wenn eine aus mehreren Schritten und damit aus mehreren Ressourcen bestehende Konfigurationsänderung mehrfach ausgeführt werden soll. Um die Ressourcen nicht jedes Mal einzeln angeben zu müssen, gibt man die zusammengesetzte Ressource an. Über die Parameter der Konfiguration, durch sie sie definiert wird, erhalten die Eigenschaften der Teil-Ressourcen ihre Werte. Ein Beispiel wäre das Einrichten eines Schulungsraums, bei der eine variable Anzahl an virtuellen Maschinen angelegt wird und zusätzliche Einstellungen vorgenommen werden müssen. In diesem Szenario werden die Ressourcen für das Anlegen der Hyper-V-VMs von Verzeichnissen mit Schulungsunterlagen und der Installation von Anwendungen in einer Ressource zusammengefasst, die dann nur noch mit der Anzahl der VMs in eine Konfiguration eingesetzt wird.

Für das Umsetzen einer zusammengesetzten Ressource gibt es keine neuen Befehlswörter. Sie wird als reguläre Konfiguration definiert, die aus mehreren Ressourcen besteht, die aber keinem Node zugeordnet werden. Damit kommt es auf die Verzeichnisstruktur an:

```
Modulname (z. B. xTrainingslabSetup)
  --Versionsnummer (z. B. 1.0)
    ----DSCResource
    ----Psd1-Datei für das Modul (z. B. TrainingslabSetup.psd1)
      ------Ressourcenmodul1 (z. B. xHyperVSetup)
      --------Psm1-Datei (z. B. xHyperVSetup.psm1)
      --------Psd1-Datei (z. B. xHyperVSetup.psd1)
```

Das Versionsverzeichnis ist optional, aber sinnvoll. Auf der obersten Ebene beziehungsweise auf der Ebene des Versionsverzeichnisses wird keine Psm1-Datei benötigt.

Die Psd1-Datei auf der obersten Ebene enthält nur ein paar Formalitäten:

```
@{
  ModuleVersion = '1.0'
  GUID = '1d5c6519-f16a-4427-a095-5b2c82cc28ba'
  Author = 'P. Monadjemi'
  Description = 'Fasst verschiedene Ressourcen für das Einrichten eines
Trainings-Labs zusammen'
 FunctionsToExport = '*'
}
```

Auch die Psd1-Datei in jedem einzelnen Modulverzeichnis, das eine Ressource definiert, aus der sich die „Oberressource“ zusammengesetzt, enthält nur ein Minimum an Einstellungen. Es kommt lediglich auf den Eintrag *RootModule* an, über den die Psm1-Datei festgelegt wird:

```
@{
  ModuleVersion = '1.0'
  RootModule = 'xHyperVSetup.schema.psm1'
  GUID = '3f76a908-3b81-4363-8808-712823391d8a'
  Author = 'P. Monadjemi'
  Description = 'Fasst verschiedene Ressourcen für das Einrichten eines
Trainings-Labs zusammen'
 FunctionsToExport = '*'
}
```

Interessant ist natürlich der Inhalt der Psm1-Datei, in diesem Beispiel „xHyperVSetup.schema.psm1“. Der folgende Auszug beschränkt sich auf den Aufbau der Datei:

```
<#
 .Synopsis
 Beispiel für eine Composite Resource
#>

configuration xHyperVSetup
{

param([Parameter(Mandatory)][ValidateNotNullOrEmpty()][String[]]$VMName,

[Parameter(Mandatory)][ValidateNotNullOrEmpty()][String]$SwitchName,
        [String]$VHDParentPath
    )

  Import-DscResource -ModuleName PSDesiredStateConfiguration
  Import-DscResource -ModuleName xHyper-V

  # Ordner für Vhd-Datei anlegen
  File VHDFolder
  {
      Ensure = "Present"
      DestinationPath = $VHDParentPath
      Type = "Directory"
  }
  # Anlegen einer Parent-Vhd
  xVHD VMVhdParent
  {
    Ensure = "Present"
    Name = "ParentVhd"
    Path = $VHDParentPath
    MaximumSizeBytes = 64GB
    Generation = "Vhd"
    DependsOn = "[File]VhdFolder"
  }

  # Jede VM anlegen
  foreach ($Name in $VMName)
  {
    # Für jede VM eine Diff-Vhd anlegen
    xVHD "VHD$Name"
    {
      Ensure = "Present"
      Name = "$Name`_Vhd"

      DependsOn = @("[File]VHDFolder")
    }

    # Jetzt die VM anlegen
    xVMHyperV "VMachine$Name"
    {
      Ensure = "Present"
      Name = $Name
      VhDPath = (Join-Path -Path $VHDParentPath -ChildPath
"$Name`_Vhd.vhd")
      SwitchName = $SwitchName
      StartupMemory = 1GB
      State = "Running"
      DependsOn = @("[xVHD]VHD$Name")
    }
  }
}
```

Befinden sich alle Dateien an ihrem Platz, steht eine Ressource mit dem Namen „xHyperV-Setup“ zur Verfügung, die wie jede andere Ressource als Teil einer Konfiguration verwendet wird. Per *Get-DSCRessource* testet man, ob die Composite-Ressource auch existiert:

```
Get-DSCResource | Where-Object ImplementedAs -eq "Composite"
```

Anschließend wird eine Konfiguration definiert, die per *Import-DSCResource* die Ressource über ihr „xTrainingsLapSetup“, zu der unter anderem die Ressource „xHyperV-Setup“ gehört, lädt und verwendet:

```
Configuration LabSetup
{
  param([String]$Computername)

Import-DSCResource-ModuleName PSDesiredStateConfiguration
Import-DSCResource-ModuleName xTrainingsLabSetup

  Node $Computername
  {
     xHyperVSetup VMSetup
     {
      VMName = @("PemoVm1", "PemoVm2")
      # Dieser Switch wird vorausgesetzt
      SwitchName = "ExternesNetzwerk"
      VHDParentPath = "E:\HyperV2"
      }
  }
}
```

Aus der Konfiguration wird wie üblich eine Mof-Datei gemacht:

```
LabSetup -Computername Localhost
```

10.9 Die PowerShell DSC-Cmdlets im Überblick

Die PowerShell fasst ab Version 5.0 die Cmdlets und Functions zum Thema DSC im Modul *PSDesiredStateConfiguration* zusammen. In Tab. 10.2 fasse ich die Commands aus diesem Modul mit einer kurzen Beschreibung zusammen. Auch wenn diese natürlich alle im Rahmen der MSDN-Dokumentation beschrieben sind, ist eine solche Kurzübersicht hilfreich, um einen Überblick über die zur Verfügung stehende Funktionalität zu erhalten. Ein Umstand, der am Anfang eventuell zu etwas Verwirrung führen kann, ist, dass das Modul auch die Ressourcen umfasst, die bei WMF von Anfang an zur Verfügung stehen.

- **Hinweis** Damit es nicht zu einfach wird, gibt es vom DSC-Team ein weiteres Modul mit dem Namen „xPSDesiredStateConfiguration“. Dahinter steckt lediglich die experimentelle Fassung des Moduls als Open Source-Projekt, das jene Ressourcen enthält, die bei WMF von Anfang an dabei sind. Die Namen aller Ressourcen beginnen daher mit einem „x“. Die „High Quality“-Version dieses Moduls heißt „PSDSCResources“. Hier geht den Namen der Ressourcen kein „x“ voraus (sie sind ja nicht experimentell).

Tab. 10.2 Die Commands aus dem Modul PSDesiredStateConfiguration

Command	Was macht es?
Disable-DscDebug	Deaktiviert den Debugmodus für Ressourcen.
Enable-DscDebug	Aktiviert den Debugmodus für Ressourcen.
Get-DscConfiguration	Gibt die DSC-Konfiguration des angegebenen Nodes aus.
Get-DscConfigurationStatus	Gibt die Informationen über die letzte oder alle bislang auf dem Node ausgeführten erfolgreichen Konfigurationen aus. Unter anderem wird ausgegeben, ob sich die einzelnen Ressourcen im „Desired State“ befinden.
Get-DscLocalConfigurationManager	Gibt die Einstellungen des LCM aus.
Get-DscResource	Gibt Details zu allen oder einer bestimmten Ressource aus. Über den Syntax-Parameter werden die Eigenschaften der Ressource ausgegeben.
New-DscChecksum	Legt die für den Pull-Modus erforderliche Prüfsummendatei für eine Ressource an.
Remove-DscConfigurationDocument	Entfernt die im Rahmen von *Start-DSCConfiguration* auf den Node übertragenen Mof-Dateien und „räumt“ damit den Node auf. Ein Remote-Node-Computer wird im Rahmen einer CIM-Session angesprochen. Der Verbose-Parameter sorgt dafür, dass sich die Aktivitäten nachvollziehen lassen.
Restore-DscConfiguration	Führt die auf einen Node zuletzt angewendete Konfiguration erneut aus. Ein Remote-Node-Computer wird im Rahmen einer CIM-Session angesprochen.
Stop-DscConfiguration	Beendet eine ausführende Konfiguration vorzeitig, indem der ausführende Job beendet wird.
Invoke-DscResource	Führt eine Ressource direkt aus, indem die Set- oder Test-Methode ausgeführt wird. Die Eigenschaften werden in Gestalt einer Hashtable übergeben. Damit lässt sich ein Konfigurationszustand (auch ohne eine Konfiguration definieren zu müssen) umsetzen.
Publish-DscConfiguration	Überträgt eine Mof-Datei direkt auf die angegebenen Node-Computer, ohne dass sie angewendet wird.
Set-DscLocalConfigurationManager	Überträgt eine Konfigurationsdatei für den LCM auf den angegebenen Node. Die Konfigurationsdatei muss sich in dem angegebenen Verzeichnis in Gestalt einer Datei mit dem Namen <NetBiosName>.meta.mof befinden.
Start-DscConfiguration	Überträgt eine oder mehrere Mof-Dateien auf die einzelnen Nodes im Push-Modus.
Test-DscConfiguration	Gibt über einen $true/$false-Wert an, ob sich der angegebene Node im „gewünschten Konfigurationszustand“ befindet oder ob dies, bedingt durch vorzeitig abgebrochene Konfigurationen oder aus anderen Gründen, nicht der Fall ist.
Update-DscConfiguration	Prüft, ob auf dem per LCM konfigurierten Pull Server eine Konfiguration wartet und wendet diese an. Ist zum Testen praktisch, da man ansonsten eine halbe Stunde warten muss, bis der LCM-Client den Pull Server kontaktiert.

10.10 Zusammenfassung

Die Themen in diesem Kapitel machen deutlich, dass es auch bei DSC trotz der anfänglich suggerierten Leichtigkeit bei der Zustandsdefinition um die typischen Herausforderungen geht, die PowerShell-Anwendern auch beim klassischen Scripting begegnen, und die einfach Zeit und Ausdauer erfordern. Insbesondere wird deutlich, dass wenn DSC im Unternehmen im größeren Stil eingesetzt werden soll, nicht nur eine Infrastruktur (Stichwort „Pull Server“) vorhanden sein muss, sondern auch eine „Manpower“ in Gestalt von Menschen erforderlich ist, die DSC-Ressourcen definieren, sich über das Verteilen von Modulen Gedanken machen und vor allem alles ausführlich testen.

Aus Text Objekte machen

11

Zusammenfassung

In diesem Kapitel geht es um ein sehr wichtiges Werkzeug für das tägliche Handwerkszeug eines PowerShell-Administrators: Es geht um die Umwandlung von Textdaten in Objekte. Warum ist diese Umwandlung so wichtig? Ganz einfach: Sobald Textdaten als Objekte vorliegen, lassen sich mit den üblichen Pipeline-Cmdlets wie *Where-Object*, *Sort-Object* oder den verschiedenen ConvertTo-Cmdlets flexibel weiterverarbeiten. Wie üblich gibt es für eine solche Umwandlung mehrere Alternativen. Ein universelles „ConvertTextTo-Object-Cmdlet" gibt es nicht, denn es hängt vom Aufbau des Textes ab, auf welche Weise dieser in Objekte mit entsprechenden Eigenschaften konvertiert werden kann. Im einfachsten Fall besteht der Text aus Zeilen mit einer festen Anzahl an Spalten, so dass er sich als Text im CSV-Format per *ConvertFrom-CSV*-Cmdlet konvertieren lässt. Dabei wird pro Zeile ein Objekt angelegt, das für jede Spalte einer Zeile eine Eigenschaft besitzt. Der Name der Eigenschaft wird entweder aus der entsprechenden Spalte der ersten Zeile genommen oder über den Header-Parameter für jede Spalte und damit für jede Eigenschaft festgelegt.

Besitzt der Text keine Unterteilung in Spalten und damit keine regelmäßige Struktur, kommt die Zerlegung des Textes über reguläre Ausdrücke in Frage. Diese Variante ist sehr flexibel und leistungsfähig, setzt aber gewisse Grundkenntnisse über reguläre Ausdrücke voraus – ein Thema, das nicht jedem behagt.

Mit dem *ConvertFrom-String*-Cmdlet, das seit der Version 5.0 der PowerShell dabei ist, gibt es eine Alternative, die ohne reguläre Ausdrücke auskommt. Das Cmdlet verwendet eine Vorlage und ist damit in der Lage, unstrukturierte Texte in Objekte zu konvertierten. Perfekt ist das Cmdlet allerdings auch nicht. Auch wenn es teilweise erstaunliche Ergebnisse liefert, ist es leider so spärlich dokumentiert, dass sich ein Erfolgserlebnis oft erst nach zähem Ausprobieren der möglichen Syntaxvarianten für die Vorlage einstellt. Es kann kleinere Wunder wirken, eine Garantie gibt es allerdings nicht.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_11

11.1 Texte im CSV-Format konvertieren

Im einfachsten Fall besteht ein Text aus Zeilen, die eine feste Aufteilung in Spalten besitzen, die durch ein einheitliches Trennzeichen voneinander getrennt sind. Dieses Textformat wird als „Comma Separated Value"-Format bezeichnet, kurz CSV. Anstelle des Kommas kann ein beliebiges Zeichen als Trennzeichen verwendet werden. Wichtig ist lediglich, dass das Trennzeichen einheitlich ist.

Der folgende Text dient als Grundlage für die nächsten Beispiele. Er besteht aus einer Reihe von Zeilen, die jeweils in drei Spalten unterteilt sind. Das Trennzeichen ist ein Semikolon.

```
Username;City;Organisation
User1;Bielefeld;IT
User2;Osnabrück;Marketing
User3;Gütersloh;Marketing
User4;Hannover;IT
```

Der Text umfasst mehrere Zeilen, die aus derselben Anzahl an Spalten bestehen. Die einzelnen Spalten sind per Semikolon getrennt. Die erste Zeile spielt eine Sonderrolle, denn sie enthält die Überschriften der einzelnen Spalten. Ein solcher strukturierter Text kann sehr einfach entweder mit dem *Import-CSV*- oder dem *ConvertFrom-CSV*-Cmdlet in Objekte konvertiert werden. Welches der beiden Cmdlets zum Einsatz kommt, hängt davon, ob sich der Text in einer Datei oder zum Beispiel in einer Variablen befindet oder von einem anderen Cmdlet geliefert wird. Bei der Konvertierung wird aus jeder Zeile ein Objekt gebildet, dessen Eigenschaften die Namen der Spaltenüberschriften erhalten. Der Wert einer Eigenschaft entspricht dem Inhalt der jeweiligen Spalte.

Für die folgenden Beispiele wird davon ausgegangen, dass sich der Text in einer Variablen mit dem Namen „Userdaten" befindet. Die Konvertierung wird daher mit dem *ConvertFrom-CSV*-Cmdlet durchgeführt.

Beispiel

Der folgende Befehl konvertiert die Textdaten in der Variablen *UserDaten* in Objekte:

```
$Userdaten | ConvertFrom-CSV
```

War die Konvertierung erfolgreich? Auf den ersten Blick sieht es so aus. Bei näherer Betrachtung stellt sich jedoch heraus, dass keine Objekte entstanden sind – die einzelnen Zeilen wurden lediglich erneut ausgegeben. Der Grund ist, dass bei *ConvertFrom-CSV* als Trennzeichen ein Komma vorausgesetzt wird. Enthält der Text ein anderes Trennzeichen, muss dieses über den *Delimiter*-Parameter angegeben werden.

Beispiel

Der folgende Befehl konvertiert erneut die Textdaten in der Variablen „UserDaten" in Objekte. Dieses Mal wird über den *Delimiter*-Parameter ein Semikolon als Trennzeichen verwendet.

```
$Userdaten | ConvertFrom-CSV -Delimiter ";"
```

11.2 Überschriften nachträglich hinzufügen oder vorhandene Überschriften ändern

Normalerweise enthält CSV-Text eine erste Zeile mit Überschriften. Diese geben die Namen der Eigenschaften der resultierenden Objekte vor. Sollte der CSV-Text keine Überschriftenzeile enthalten oder sollen die Eigenschaften andere Namen erhalten als die Namen in der Überschriftenzeile, ist dies kein Problem. Dafür gibt es sowohl beim *Import-CSV*- als auch beim *ConvertFrom-CSV*-Cmdlet den Header-Parameter, über den die Namen der Spaltenköpfe (nachträglich) festgelegt werden – und damit auch jene Namen für die einzelnen Eigenschaften, die jedes Objekt besitzt, das eine Zeile repräsentiert.

Beispiel

Der folgende Befehl konvertiert CSV-Text in Objekte und legt die Namen der drei Eigenschaften fest.

```
$UserDaten | ConvertFrom-CSV -Delimiter ";" -Header "Benutzername",
"Stadt", "Firma"
```

Gab es im CSV-Text bereits eine Überschriftenzeile, wird diese durch den Header-Parameter zu einer regulären Zeile. Damit sie nicht weiterverarbeitet wird, muss ein *Select-Object*-Cmdlet mit dem *Skip*-Parameter angehängt werden, der den Wert 1 erhält.

Beispiel

Der folgende Befehl konvertiert CSV-Text in Objekte und legt per Header-Parameter die Namen der Eigenschaften fest. Da das erste Objekt aus der Überschriftenzeile entstanden ist, wird es bei der Ausgabe durch ein *Select-Objec*t-Cmdlet über dessen Skip-Parameter ausgelassen.

```
$UserDaten | ConvertFrom-CSV -Delimiter ";" -Header "Benutzername",
"Stadt", "Firma" | Select-Object –Skip 1
```

11.3 Unregelmäßige Texte zerlegen

Besitzt ein Text keinen regelmäßigen Aufbau, gibt es grundsätzlich zwei Möglichkeiten, ihn in Objekte umzuwandeln. Mit Hilfe des Split-Operators beziehungsweise mit Hilfe jener String-Funktionen, die jede Zeichenkette als Methoden-Members besitzt. Oder mit Hilfe regulärer Ausdrücke und dem *Select-String*-Cmdlet, dem *Match*-Operator oder dem Typenalias *[Regex]*, der die Methoden *Match* und *Matches* zur Verfügung stellt.

11.3.1 Texte mit dem Split-Operator zerlegen

Der leistungsfähige *Split*-Operator zerlegt einen Text anhand eines Trennzeichens, eines Scriptblocks oder eines regulären Ausdrucks. Er besitzt zahlreiche Optionen, die in der Hilfe unter „about_split“ ausführlich beschrieben werden.

Beispiel

Im folgenden Beispiel wird der Inhalt einer Datei *ADUserdaten.txt* mit Hilfe des *Split*-Operators in ihre Bestandteile zerlegt. Die Datei besitzt den folgenden Aufbau:

```
User1,Itzehohe/7.6.2016-50
User2,Ippenbühren/12.8.2016-24
User3,Ingoldstadt/25.6.2015-7
User4,Ingersheim/12.5.2015-7
```

Da jede Zeile mit dem Komma, dem Schrägstrich und dem Minuszeichen insgesamt drei Trennzeichen enthält, lässt sich der Text nicht mehr mit dem *ConvertFrom-CSV*-Cmdlet konvertieren.

Im ersten Aufruf wird der Text lediglich mit dem Trennzeichen `n, das für ein Zeilenumbruchszeichen steht, in mehrere Zeilen unterteilt. Damit der Operator nicht als Parameter des *Get-Content*-Cmdlets interpretiert wird, muss das Cmdlet in runde Klammern gesetzt werden:

```
(Get-Content –Path ADUserDaten.txt) –split "`n"
```

Das Ergebnis ist nicht gerade prickelnd, da lediglich die einzelnen Zeilen untereinander ausgegeben werden. Doch der *Split*-Operator hat zum Glück noch etwas mehr zu bieten.

Soll ein Text mit mehreren unterschiedlichen Trennzeichen pro Zeile zerlegt werden, gibt es mehrere Möglichkeiten, um das Ziel zu erreichen. Zum Beispiel könnte man mehrere *Split*-Operatoren aneinanderreihen. Etwas eleganter ist die Option, anstelle eines einzelnen Trennzeichens einen Scriptblock zu übergeben, bei dem jedes Zeichen über die Variable *$_* angesprochen wird. Liefert der Scriptblock einen *$true*-Wert, wird die Zeile bei diesem Zeichen getrennt. In dem Scriptblock müssen daher lediglich die einzelnen Trennzeichen per *eq*-Operator verglichen werden.

Beispiel

Der folgende Aufruf des *Split*-Operators trennt jede Zeile der Datei *ADUserDaten.txt* anhand mehrerer Trennzeichen in ihre Bestandteile.

```
(Get-Content -Path $TmpPfad -Encoding Default) -split { $_ -eq "," -or $_
-eq "/" -or $_ -eq "-"}
```

Pro Zeile liefert der *Split*-Operator vier Textelemente.

Beispiel

Das folgende Beispiel ist bereits etwas umfangreicher, da aus den Textelementen, die der *Split*-Operator pro Zeile liefert, ein Objekt gemacht wird. Dabei wird die Konvertierung mit dem *PSCustomObject*-Typ und einer Hashtable durchgeführt, über die die Eigenschaften des Objekts mit ihren Werten definiert werden. Durch die Umwandlung wird jede Zeile in ein Objekt umgewandelt, das die Eigenschaften *Username*, *Ort*, *Datum* und *Tage* besitzt.

```
<#
 .Synopsis
 Textzeilen mit mehreren Trennzeichen in Objekte konvertieren
#>

$Mustertext = @"
User1,Itzehohe/7.6.2016-50
User2,Ippenbühren/12.8.2016-2
User3,Ingoldstadt/25.6.2015-99
User4,Ingersheim/12.5.2015-123
"@

$Elemente = ($Mustertext -split "`r`n") -split {  $_ -eq "," -or $_ -eq
"/" -or $_ -eq "-"}

for($i = 0; $i -le $Elemente.Count / 4; $i++)
{
    [PSCustomObject]@{
        Name=$Elemente[$i*4+0]
        Ort=$Elemente[$i*4+1]
        Datum=$Elemente[$i*4+2]
        Nr=$Elemente[$i*4+3]
    }
}
```

Beim Zerlegen von Zeilen aus einer Textdatei kommt es darauf an, dass jede Zeile in der Regel mit zwei Sonderzeichen abgeschlossen wird: Einem Carriage Return-Zeichen (ASCII-Code 13) und einem Line Feed-Zeichen (ASCII-Code 10). Beide Zeichen müssen beim Zerlegen der Zeile „irgendwie“ ausgefiltert werden, da sie ansonsten Teil des letzten Zeilenelements sind. Auch das ist mit dem *Split*-Operator elegant möglich, da auch mehrere Trennzeichen folgen können. Ein

```
$Mustertext -split "`r`n"
```

zerlegt einen mehrzeiligen Text in die einzelnen Zeilen ohne die beiden Zeilenabschlusszeichen.

Neben dem *Split*-Operator gibt es bei der PowerShell einen zweiten „Zeichenkettentrenner“: Die *Split*-Methode, die jede Zeichenkette besitzt. Auch wenn sich beide auf den ersten Blick sehr ähnlich sind, gibt es einen wichtigen Unterschied. Der *Split*-Operator ist deutlich leistungsfähiger, da er u. a. reguläre Ausdrücke verwendet. Sie sollten daher diesem Operator den Vorzug geben.

11.3.2 Texte mit Hilfe regulärer Ausdrücke zerlegen

Die flexibelste Alternative für das Zerlegen von Texten sind reguläre Ausdrücke, auch „Regexe“ genannt. Ein regulärer Ausdruck ist ein Muster, das mit Hilfe einer eigenen Syntax zusammengestellt wird. Bei Anwenden des Musters auf einen Text werden alle Zeilen ausgegeben, die eine Stelle enthalten, die mit dem Muster übereinstimmt. Reguläre Ausdrücke werden bei der PowerShell beim *Select-String*-Cmdlet, beim *Match*-Operator und beim *[Regex]*-Typenalias verwendet.

▶ **Tipp** Die PowerShell-Hilfe gibt unter „about_regular_expressions“ eine gute Übersicht über den Umgang mit regulären Ausdrücken.

Beispiel

Ausgangspunkt für das nächste Beispiel ist eine Liste mit Namen, die aus Buchstaben, Zahlen und Sonderzeichen bestehen kann.

```
Server123
PC-077
Rack[99]
Server/489
PC456
Server_A100
```

Die Aufgabe besteht darin, Namen und Zahlen sauber zu trennen, so dass jeder Name in zwei Teile zerlegt wird – die Sonderzeichen zwischen den beiden Namensteilen sollen dabei keine Rolle spielen. Was sich zunächst einfach anhört, wird durch den Umstand erschwert, dass sich zwischen den Namen und den Ziffern verschiedene Sonderzeichen befinden können, aber nicht müssen. Auch wenn es natürlich möglich ist, das Zerlegen mit den String-Methoden durchzuführen, einfacher geht es mit Hilfe eines regulären Ausdrucks.

Das Muster besitzt den folgenden Aufbau:

```
"^([a-z]+)[-[/_]*([a-z0-9]+)"
```

Das Wichtigste an dem Muster sind die runden Klammern, da durch sie Gruppen gebildet werden, über die die gefundenen Übereinstimmungen später über einen Index oder einen Namen abgerufen werden. Wird kein Name vergeben, wird jede Gruppe über einen Index angesprochen, der bei 1 beginnt. Der Index 0 steht für den gesamten Treffer.

Beispiel

Die folgende Befehlsfolge zerlegt den Text mit den Namen in einzelne Zeilen und zerlegt jede Zeile mit Hilfe des regulären Ausdrucks und dem *Match*-Operator in zwei Teile: Name und Nummer.

```
Muster = "^([a-z]+)[-[/_]*([a-z0-9]+)"

foreach($Name in $Namen -split "`n")
{
  [void]($Name -match $Muster)
  [PSCustomObject]@{
                      Server = $Matches[1]
                      Id = $Matches[2]
                    }
}
```

Auch in diesem Beispiel wird mit dem Typenalias *[PSCustomObject]* ein Objekt gebildet, das die Bestandteile einer Zeile als Eigenschaften anbietet.

Das *Select-String*-Cmdlet liefert pro Treffer ein *MatchInfo*-Objekt.

Der folgende Befehl zerlegt die Servernamen mit dem *Select-String*-Cmdlet in ihre Bestandteile:

```
$ServerNamen -split "`n" | Select-String -Pattern $Muster | Select
@{n="Name";e={$_.Matches[0].Groups[1]}},
@{n="Nr";e={$_.Matches[0].Groups[2]}}
```

Die dritte Option für das Zerlegen von Texten mit Hilfe regulärer Ausdrücke stellt der *[Regex]*-Typenalias dar, der unter anderem die statischen Methoden *Match* und *Matches* anbietet. Während die *Match*-Methode nur den ersten Treffer als *Match*-Objekt zurückgibt, gibt die *Matches*-Methode alle Treffer zurück.

Beispiel

Das folgende Beispiel holt über die *Matches*-Methode alle Treffer.

```
[Regex]::Matches($ServerNamen, $Muster)
```

Obwohl das *Select-String*-Cmdlet einen *AllMatches*-Parameter besitzt, ist die *Matches*-Methode der einfachste Weg, um alle Treffer über einen Regex zu erhalten.

11.4 Objekte anlegen

Das Zerlegen von Texten ist nur der erste Schritt. Für eine komfortable Weiterverarbeitung muss das Ergebnis in Objekte konvertiert werden. Ein Beispiel ist der *Split*-Operator, der mit Hilfe eines regulären Ausdrucks einen Text in seine Bestandteile zerlegt. Damit sich diese flexibel weiterverarbeiten lassen, müssen aus den Textfragmenten Objekte gemacht werden.

Objekte können auf mehrere Art und Weisen angelegt werden: Über das *New-Object*-Cmdlet, über die Konvertierung einer Hashtable mit dem *PsCustomObject*-Typenalias oder neuerdings über die statische *New()*-Methode eines Typobjekts. Die erste Variante muss immer dann angewendet werden, wenn dem neuen Objekt beim Anlegen Werte über den Konstruktor der Klasse übergeben werden sollen. Die zweite Variante ist immer dann praktischer, wenn nur ein Objekt mit ein paar Eigenschaften und Werten entstehen soll. Die dritte Variante ist insofern praktisch, da sie am wenigsten Tippaufwand erfordert.

11.4.1 Objekte mit dem New-Object-Cmdlet anlegen

Das *New-Object*-Cmdlet legt ein neues Objekt an. Als einzige Angabe wird der Name des Typs benötigt, auf dem das Objekt basieren soll. Der Typ (eine andere Bezeichnung ist Klasse) legt fest, welche Members das neue Objekt besitzt. Soll ein bezüglich seiner Members leeres Objekt angelegt werden, wird dafür der Typ *PsCustomObject* verwendet

(alternativ kommt auch der Typenalias *PsObject* in Frage). Dies ist der allgemeinste Typ, den die PowerShell zu bieten hat.

Die Eigenschaften des neuen Objekts werden beim *New-Object*-Cmdlet über den Property-Parameter und eine Hashtable angegeben. Die Hashtable enthält eine Liste von Name=Wert-Paaren für die einzelnen Eigenschaften.

Beispiel

Der folgende Befehl legt ein Objekt an, das mit *P1* und *P2* zwei Eigenschaften besitzt, die als Werte 100 und 200 erhalten.

```
New-Object -Typename PSCustomObject -Property @{P1 = 100; P2 = 200 }
```

11.4.2 Objekte mit einer Hashtable anlegen

Etwas einfacher geht das Anlegen eines neuen Objekts mit dem *[PSCustomObject]*-Typenalias und einer Hashtable.

Beispiel

Der folgende Befehl legt über ein Hashtable ein Objekt an, das mit *P1* und *P2* zwei Eigenschaften erhält, die als Werte 100 und 200 besitzen.

```
[PsCustomObject]@{P1 = 100; P2 = 200 }
```

11.4.3 Unregelmäßige Textdaten mit dem ConvertFrom-String-Cmdlet verarbeiten

Das *ConvertFrom-String*-Cmdlet wurde erst mit der Version 5.0 der PowerShell eingeführt. Es konvertiert Texte mit einer unregelmäßigen Struktur, die nicht aus einer festen Anordnung von Zeilen und Spalten besteht, in Objekte. Anstelle von regulären Ausdrücken arbeitet das Cmdlet mit Textvorlagen, die den Aufbau des zu zerlegenden Textes beschreiben. Die Textstellen, auf die es ankommt, werden in geschweifte Klammern gesetzt, in denen der Name einer Eigenschaft eingetragen ist. Diese Werte werden in Eigenschaften des resultierenden Objekts umgesetzt. Das Cmdlet lernt anhand der Vorlage den Aufbau des Textes. In der Regel genügen ein paar Zeilen, um die Struktur eines Textes zu beschreiben, der aus einigen tausend Zeilen bestehen kann. Klassische Beispiele sind die Ausgaben von Befehlszeilentools, bei denen sich die zu verarbeitenden Daten mitten im Text befinden.

Beispiel

Das folgende Beispiel geht von einem Text aus, in dem sich Daten mit allgemeinen Angaben abwechseln. Das Ziel ist es, die Daten aus diesem Text zu extrahieren und daraus Objekte zu machen.

```
  Daten
  ==================
  Wert1: 1000
  Wert2: 2000
  Wert3: 3000

  ==================
  Wert1: 1001
  Wert2: 2001
  Wert3: 3001
  ==================
  Wert1: 1002
  Wert2: 2002
  Wert3: 3002
  ==================
  Wert1: 1003
  Wert2: 2003
  Wert3: 3003
```

Auch wenn sich der Text mit vertretbarem Aufwand mit Hilfe von String-Verarbeitung oder regulären Ausdrücken zerlegen ließe, deutlich eleganter geht es mit dem *Convert-From-String*-Cmdlet.

Dazu wird als erstes eine Vorlage benötigt:

```
$Vorlage = @'
  Daten
  ==================
  Wert1: {W1*:1234}
  Wert2: {W2:1234}
  Wert3: {W3:1234}
,  ==================
  Wert1: {W1*:1234}
  Wert2: {W2:1234}
'@
```

Anschließend wird das *ConvertFrom-String*-Cmdlet ausgeführt.

```
$Textdaten | ConvertFrom-String -TemplateContent $Vorlage
```

Das Ergebnis sind Objekte mit einer *W1*- und einer *W2*-Eigenschaft, die die einzelnen Werte von „Wert1“ und „Wert2“ enthalten.

Auch wenn das Ergebnis mit Sicherheit beeindruckend ist, so einfach geht es leider nicht immer. Es genügen schon kleine Änderungen im Text oder in der Vorlage, um eine genauso umfangreiche wie nichtssagende Fehlermeldung zu produzieren. Immerhin bietet das Entwicklerteam in jeder Fehlermeldung an, dass man ihm die Textprobe per E-Mail senden kann – die Adresse ist in der Fehlermeldung angegeben. Die ganze Technik befindet sich daher noch in der Umsetzungsphase.

11.4.4 XML-Daten verarbeiten

Eine der Stärken der PowerShell ist der Umgang mit XML-Daten. XML-Daten lassen sich dank des *[Xml]*-Typenalias einfach einlesen und als Objekte weiterverarbeiten. Dabei greift Ihnen die PowerShell mit seinem Typenanpassungssystem unter die Arme, indem es

die Unterelemente und Attribute als (echte) Eigenschaften an das resultierende *XmlDocument*-Objekt hängt. Damit wird aus einer XML-Struktur ein Objekt mit Eigenschaften, deren Namen und Werte den Namen von Elementen und ihren Inhalten und den Namen von Attributen und ihren Werten entsprechen.

Beispiel

Zur Einstimmung ein kleines Beispiel. Der folgende XML-Text ist sehr einfach gestrickt.

```
$XmlDaten = @'
<root>
 <el a='1'>100</el>
 <el a='2'>200</el>
 <el a='3'>300</el>
</root>
'@
```

Der XML-Text besteht aus einem Stammelement *root* und einer Reihe von *el*-Elementen, die jeweils ein Attribut a besitzen. Jedes *el*-Element besitzt zudem einen Textinhalt (#Text).

Beispiel

Der folgende Befehl macht aus dem XML-Text ein Objekt, über dessen Eigenschaften die einzelnen Elemente angesprochen werden.

```
Daten = ([Xml]$XmlDaten)
```

Über die *root*-Eigenschaft wird das gleichnamige Stammelement angesprochen. Das resultierende Objekt besitzt eine Eigenschaft *el*. Über sie werden alle *el*-Elemente angesprochen. Jedes daraus resultierende Objekt besitzt zwei Eigenschaften: *a* und *#text*. Damit wird der XML-Text dank dem *[Xml]*-Typenalias „mundgerecht“ serviert.

Beispiel

Der folgende Befehl gibt die Inhalte aller *el*-Elemente aus.

```
$Daten.root.el."#text"
```

Auch Änderungen sind natürlich möglich.

Beispiel

Der folgende Befehl verdoppelt die Inhalte aller *el*-Elemente.

```
$Daten.root.el.ForEach{ $_."#text" = [String]([int]$_."#text" * 2)}
```

Die etwas umständliche Hin und Her-Datentypkonvertierung ist erforderlich, da für die Berechnung aus dem Text ein *Int* gemacht werden muss, dem Element aber nur ein String zugewiesen werden darf, so dass das Ergebnis der Multiplikation wieder in einen String konvertiert werden muss.

Für die Speicherung des aktualisierten XMLs in einer Datei sorgt die *Save()*-Methode des *XmlDocument*-Objekts.

Beispiel

Der folgende Befehl speichert den XML-Text, der in der Variablen *Daten* enthalten ist.

```
$Daten.Save((Join-Path (gl) "XmlDatenNeu.xml"))
```

Da die *Save()*-Methode nicht automatisch das aktuelle Verzeichnis berücksichtigt, muss der Pfad vor dem Abspeichern aus dem aktuellen Verzeichnispfad und dem Dateinamen per *Join-Path*-Cmdlet gebildet werden.

Eine Alternative zur automatischen XML-Konvertierung durch das Typensystem der PowerShell ist der Zugriff auf das XML über *XPath* mit dem *Select-Xml*-Cmdlet.

Beispiel

Der folgende Aufruf des *Select-Xml*-Cmdlets gibt die Attribute und Werte der einzelnen *el*-Elemente als Objekte aus.

```
Select-Xml -Content $XmlDaten -XPath "//el" | ForEach {
[PSCustomObject]@{Attrib=$_.Node.Attributes["a"].Value;
Wert=$_.Node.InnerText} }
```

Enthält das XML Namespace-Deklarationen und Namespace-Präfixe, müssen diese über den *Namespace*-Parameter von *Select-Xml* und einer Hashtable, in der dem Präfix der Namespace zugeordnet wird, berücksichtigt werden.

Beispiel

Das folgende Beispiel verwendet die XML-Daten aus den letzten Beispielen, nur dass die *el*-Elemente mit einem Namespace-Präfix erweitert wurden.

```
$XmlDaten = @'
<root xmlns:test="urn:xyz">
 <test:el a='1'>100</test:el>
 <test:el a='2'>200</test:el>
 <test:el a='3'>300</test:el>
</root>
'@
```

Für den Zugriff auf die *el*-Elemente wird eine Hashtable benötigt:

```
$Ns = @{test="urn:xyz"}
```

Der folgende Aufruf des *Select-Xml*-Cmdlets verwendet diese Hashtable und stellt dem *el*-Element sein Namespace-Präfix voraus.

```
Select-Xml -Namespace $Ns -Content $XmlDaten -XPath "//test:el" | ForEach
{ [PSCustomObject]@{Attrib=$_.Node.Attributes["a"].Value;
Wert=$_.Node.InnerText}}
```

Einfacher geht das Einlesen mit dem Typenalias, dem Präfixe und andere Formalitäten egal sind.

Auch wenn der Verarbeitungsaufwand größer ist als bei der automatischen Typenkonvertierung, insgesamt ist die XPath-Variante die flexiblere Alternative. Eine Ausnahme sind XML-Daten mit Namespace-Deklarationen und Elementen, deren Namen ein Namespace-Präfix vorausgeht. Hier ist die Typenanpassung per *[Xml]* im Vorteil, da Namespace-Präfixe einfach ignoriert werden, während sie bei XPath entsprechend berücksichtigt werden müssen.

Beispiel

Zum Abschluss dieses Themenblocks folgt ein weiteres Beispiel für die Verarbeitung von XML-Daten, das etwas praxisnäher ist. Es handelt sich um Daten, die aus einer Hard- und Software-Inventur stammen. Das Ziel soll es sein, die Summe des Preis-Attributs zu ermitteln, was mit einem relativ einfachen Befehl möglich sein wird.

```
$XmlDaten = @'
 <inventar>
   <geraet id="1000" preis="100">
     <kategorie>Drucker</kategorie>
     <bezeichnung>HP Laserjet II</bezeichnung>
   </geraet>
   <geraet id="1001" preis="120">
     <kategorie>Drucker</kategorie>
     <bezeichnung>Epson Fx80</bezeichnung>
   </geraet>
   <geraet id="1002" preis="40.50">
     <kategorie>Festplatte</kategorie>
     <bezeichnung>Seagate 200MByte</bezeichnung>
   </geraet>
 </inventar>
'@
```

Der folgende Befehl berechnet mit Hilfe des *Measure-Object*-Cmdlets die Summe der Werte des Attributs *Preis*.

```
(([Xml]$XmlDaten).inventar.geraet | Measure-Object -Property Preis -
Sum).Sum
```

11.5 JSON-Daten verarbeiten

Das *JavaScript Object Notation*-Format, kurz JSON, spielt in erster Linie beim Aufruf von Webservice-Funktionen eine Rolle. Die PowerShell bietet für das Verarbeiten von JSON-Daten das *ConvertFrom-JSON*-Cmdlet und für das Konvertieren von Objekten in das JSON-Format das *ConvertTo-JSON*-Cmdlet.

Beispiel

Der folgende Befehl wählt ein paar Process-Objekte mit drei Eigenschaften aus und wandelt die Objekte in das JSON-Format um.

```
Get-Process | Where WS -gt 50MB | Select Name,WS,StartTime | ConvertTo-
JSON
```

Das Ergebnis besitzt den folgenden Aufbau:

```
[
  {
    "Name":  "dns",
    "WS":  128868352,
    "StartTime":  "\/Date(1474532425772)\/"
  },
  {
    "Name":  "lsass",
    "WS":  59416576,
    "StartTime":  "\/Date(1474532411994)\/"
  }
]
```

Beispiel

Der folgende Befehl führt eine Suche gegen die öffentliche Musikdatenbank iTunes der Firma Apple durch und wandelt die zurückgegebenen Textdaten im JSON-Format mit dem *ConvertFrom-JSON*-Cmdlet in Objekte um, die über die Eigenschaften *ArtistName*, *CollectionName* und *TrackName* verfügen.

```
$SearchTerm = "Bruce Springsteen"
$Uri = https://itunes.apple.com/search?term=$SearchTerm
$Result = Invoke-WebRequest -Uri $Uri
$Result.Content | ConvertFrom-Json | Select-Object -ExpandProperty
results | Select-Object ArtistName, collectionName, trackName
```

11.6 Zusammenfassung

Objekte sind bei der PowerShell die bevorzugte Form der Ausgabe. Cmdlets geben grundsätzlich Objekte aus. Soll zum Beispiel im Rahmen eines Skripts eine individuelle Ausgabe erfolgen, wird diese entweder mit dem *New-Object*-Cmdlet oder mit dem *PsCustomObject*-Typenalias erzeugt. Die Eigenschaften des neuen Objekts werden entweder über den Property-Parameter oder über eine Hashtable festgelegt. Der Vorteil von Objekten als Alternative zur Textausgabe ist, dass sich die Ausgabe mit den dafür vorgesehenen Cmdlets wie *Select-Object* oder *Sort-Object* weiterverarbeiten lässt.

Active Directory-Administration

12

Zusammenfassung

In diesem Kapitel lernen Sie die Administration eines Active Directory mit Hilfe der Cmdlets des ActiveDirectory-Moduls kennen. Die Themenauswahl reicht von einfachen Abfragen von Benutzerkonten über Änderungen an Benutzerkonten bis zum Einrichten einer Domäne und eines Domänenkontrollers. Seit Windows Server 2012 ist Dcpromo in Rente geschickt worden, so dass die PowerShell-Cmdlets die einzige Option darstellen, um diese Aufgabe in der Befehlszeile ausführen zu können. Als kleinen „Bonustrack" zeige ich am Ende, dass sich auch Open LDAP-Server relativ problemlos ansprechen lassen, wenngleich hier nicht der Komfort von Cmdlets zur Verfügung, sondern die Directory-Funktionen der .NET-Laufzeit mit Hilfe von vordefinierten Typenalias angesprochen werden.

Wenn es ein Fazit gibt, dann dass die PowerShell das ideale Werkzeug für die Administration eines Active Directorys über Skripte und die Befehlszeile ist. Im Mittelpunkt steht das *ActiveDirectory*-Modul. Es ist bei der PowerShell allerdings nicht von Anfang an dabei, sondern ein Teil von Windows Server, das gegebenenfalls als Feature hinzugefügt werden muss. Für Windows 7 und aufwärts steht das Modul im Rahmen der *Remote Server Administration Tools* (RSAT) zur Verfügung, die nachträglich hinzugefügt werden müssen.

► **Hinweis** Die Version der PowerShell spielt für das ActiveDirectory-Modul keine Rolle. Es kommt jede Version ab 2.0 in Frage.

Beispiel

Der folgende Befehl fügt das ActiveDirectory-Modul unter Windows Server hinzu.

```
Install-WindowsFeature –Name RSAT-AD-PowerShell
```


P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_12

Das ActiveDirectory-Modul wurde mit Windows Server 2008 R2 eingeführt. In der ersten Version umfasste es lediglich 76 Cmdlets, bei Windows Server 2016 sind es 147 Cmdlets. Damit lassen sich alle Bereiche eines Active Directorys ansprechen.

▶ **Hinweis** Das ActiveDirectory-Modul ist nicht die einzige Option für die AD-Verwaltung. Vor der Einführung des Moduls war mit den Quest-AD-Cmdlets ein Satz von Cmdlets, die von der Firma Quest herausgegeben wurden, weit verbreitet. Über die Typenalias *[ADSI]* und *[ADSISearcher]* lassen sich sowohl ein Active Directory-Verzeichnis als auch allgemein LDAP-Verzeichnisse direkt ansprechen. Damit sind einfache Aktionen wie das Anlegen oder Entfernen von Benutzerkonten und Abfragen auf der Basis von LDAP möglich. Eine weitere Alternative ist, die Befehlszeilentools von Windows Server wie *dsget* oder *dsquery* oder ein Tool wie *ADFind* auszuführen. Dies sind aber in erster Linie nur Not- und Nischenlösungen. Der beste Weg, um das AD per PowerShell zu administrieren, ist die Verwendung des ActiveDirectory-Moduls.

12.1 AD DS und Domänencontroller einrichten

Ein Active Directory inklusive Domänencontroller kann per PowerShell-Cmdlets mit wenig Aufwand eingerichtet werden. Das langjährige Tool *DCPromo.exe* wurde bereits mit Windows Server 2012 in den Ruhestand geschickt und durch das *Install-ADDSDomainController*-Cmdlet abgelöst.

Beispiel

Der folgende Befehl fügt das Feature „AD-Domain-Services“ inklusive der Verwaltungstools hinzu.

```
Install-WindowsFeature –Name AD-Domain-Services –IncludeManagementTools -
IncludeAllSubFeatures
```

Das Einrichten eines Domänencontrollers übernehmen die Cmdlets *Install-ADDSDomainController* und *Install-ADDSForest* aus dem Modul *ADDSDeployment. Install-ADDSForest* muss für den ersten Domänencontroller, *Install-ADDSDomainController* für jeden weiteren Domänencontroller ausgeführt werden.

Beispiel

Der folgende Aufruf richtet einen Domänencontroller für die Domäne „pskurs.local“ ein.

```
$PwSec = "demo+123" | ConvertTo-SecureString –AsPlainText –Force
Install-ADDSForest `
 -SkipPreChecks `
 -DomainName pskurs.local `
 -SafeModeAdministratorPassword $PwSec `
‘ -DomainMode Win2012R2 `
 -LogPath $env:documents\ADDSInstall.log `
 -Force
```

▶ **Tipp** Ob für das Einrichten einer Domäne beziehungsweise eines Domänencontrollers alle Voraussetzungen erfüllt sind, kann über die Cmdlets *Test-ADDSDomainInstallation* und *Test-ADDSDomainControllerInstallation* vor einer Installation getestet werden.

12.1.1 Aktualisieren der Hilfe

Nach dem Hinzufügen des ActiveDirectory-Moduls sollte die Hilfe per *Update-Help*-Cmdlet aktualisiert werden. Die Hilfe zu dem Modul ist umfangreich und umfasst mehrere Themen:

```
>about_ActiveDirectory
>about_ActiveDirectory_Filter
>about_ActiveDirectory_Identity
>about_ActiveDirectory_ObjectModel
```

▶ **Hinweis** Im Unterschied zu allen anderen Tools (WSH, AdFind usw.) basiert das ActiveDirectory-Modul nicht auf ADSI, sondern auf Ws-Management und damit auf Webservice-Aufrufen. Es sind auf dem Domänencontroller deswegen aber keinerlei Vorbereitungen erforderlich, insbesondere muss kein PowerShell-Remoting aktiviert werden. Auch die PowerShell muss auf dem Domänencontroller nicht vorhanden sein. Der Datenaustausch findet über den Port 9389 statt (das kann zum Beispiel für den Zugriff auf eine Azure VM ein Thema sein). Eine Übersicht über alle involvierten Ports gibt es unter der folgenden Adresse: http://technet.microsoft.com/de-de/library/dd772723%28v=ws.10%29.aspx.

12.1.2 Authentifizierung

Die Authentifizierung erfolgt über Kerberos, das heißt mit der Windows-Anmeldung sind alle Voraussetzungen für den Zugriff auf das Active Directory (AD) geschaffen. Eine separate Authentifizierung ist daher nicht erforderlich. Werden die Active Directory-Cmdlets (AD-Cmdlets) auf einem Computer ausgeführt, der nicht Mitglied einer Domäne ist, muss der Domänencontroller per *Server*-Parameter verwendet werden, über den jedes AD-Cmdlet verfügt. Die Authentifizierung erfolgt wie üblich über den *Credential*-Parameter. Die Möglichkeiten, die Ihnen nach der Anmeldung zur Verfügung stehen, werden durch das Benutzerkonto (einfacher Benutzer, Domänadministrator usw. bestimmt).

▶ **Hinweis** Sollte beim Importieren des ActiveDirectory-Moduls eine Warnung erscheinen, liegt dies daran, dass kein Domänencontroller zur Anmeldung gefunden wurde.

12.2 Suche nach Benutzerkonten per Get-AdUser

Das *Get-ADUser*-Cmdlet führt eine Suche im gesamten Verzeichnis durch, das auch aus mehreren Verzeichnissen bestehen kann. Der Suchfilter wird entweder per PowerShell-Filtersyntax mit dem *Filter*-Parameter oder über einen LDAP-Filter mit dem *LDAPFilter*-Parameter festgelegt.

Beispiel

Der folgende Befehl holt alle Benutzerkonten, deren Name mit „A“ beginnt.

```
Get-ADUser –Filter { Name –like "A*" }
```

Die folgende Abfrage verwendet anstelle eines PowerShell-Filters einen Standard-LDAP-Filter.

```
Get-Aduser -LDAPFilter "(&(ObjectClass=user)(l=Esslingen))"  -Properties
City
```

12.3 Die PowerShell-Abfragesyntax

Das ActiveDirectory-Modul bietet eine eigene Abfragesyntax, die stark an die PowerShell-Syntax angelehnt ist. Bei dieser Syntax gibt es aber ein paar Unterschiede:

- Es gibt keinen Match-Operator
- Als Platzhalter gibt es nur den *
- Vergleiche mit Ausdrücken sind nicht möglich
- Vergleiche mit $null sind nicht möglich
- Es gibt einen Approx-Operator als Pendant zu ~= bei LDAP
- Es gibt einen RecursiveMatch-Operator

Beispiel

Die folgende Abfrage gibt alle Benutzerkonten zurück, die sich nach einem bestimmten Stichtag nicht mehr angemeldet haben. Für den Vergleich muss eine Variable verwendet werden, da ein direkter Vergleich mit einem Ausdruck nicht möglich ist.

```
$Datum = Get-Date –Date "1.1.2016"
Get-ADUser –Filter { Name –like "A*" -and LastLogonDate –lt $Datum }
```

12.3.1 Benutzerkonten auswählen über den Identity-Parameter

Per *Get-ADUser* kann ein Benutzerkonto auch direkt geholt werden. In diesem Fall kommt der *Identity*-Parameter ins Spiel, auf den zum Beispiel der Name oder der DN des Benutzerkontos folgt.

Beispiel

Der folgende Befehl holt das Administratorkonto über seinen Namen.

```
Get-ADUser –Identity Administrator
```

Beispiel

Der folgende Befehl holt das Administratorkonto über seinen Distinguished Name (DN).

```
Get-ADUser –Identity
"CN=Administrator,CN=Users,DC=mobil,DC=pshub,DC=local"
```

12.3.2 Die Rolle des Properties-Parameter bei Get-ADUser

Per Voreinstellung geben Cmdlets wie *Get-ADUser* nur Objekte mit zehn Attributen zurück: *DistinguishedName*, *Enabled*, *GivenName*, *Name*, *ObjectClass*, *ObjectGUID*, *SamAccountName*, *SID*, *Surname* und *UserPrincipalName*. Alle weiteren Attribute müssen über den *Properties*-Parameter ausgewählt werden. Der Parameter unterstützt keine Platzhalter. Es werden entweder die Namen der Attribute oder ein * angegeben, um sämtliche Attribute zu erhalten.

Der folgende Befehl holt das Attribut *EMailAddress*:

```
Get-AdUser –Identity Administrator –Properties EMailAddress
```

Da „EMailAddress“ kein offizielles LDAP-Attribut ist, funktioniert der Aufruf natürlich auch, wenn der Originalname, in diesem Fall „mail“ angegeben wird.

Beispiel

Der folgende Befehl holt ebenfalls das E-Mail-Attribut.

```
Get-AdUser –Identity Administrator –Properties Mail
```

► **Tipp** Möchte man generell wissen, welche Attribute zur Auswahl stehen, erfährt man dies, indem man alle Attribute holt und per *Get-Member* nur jene übrig lässt, die einen aktuell interessieren.

Beispiel

Der folgende Befehl gibt alle Attribute aus, in deren Namen das Wort „Logon“ enthalten ist.

```
Get-AdUser –Identity Administrator –Properties * | Get-Member –Name
*Logon*
```

12.3.3 Spezialfall zusammengesetzte Attribute

Zusammengesetzte Attribute („constructed attributes“) können nicht mit Platzhaltern in einen Filterausdruck eingebaut werden. Beispiele für solche Attribute sind *Distinguished-Name*, *Member* und *MemberOf*.

Beispiel

Die folgende Abfrage führt aufgrund der Platzhalter zu keinem Ergebnis, auch wenn sie logisch erscheint.

```
Get-AdUser -Filter { DistinguishedName -like "*Verwaltung*" }
```

Damit die Abfrage ausgeführt werden kann, muss das Attribut mit einem konkreten Wert verglichen werden.

```
Get-AdUser -Filter { DistinguishedName -eq
"OU=Verwaltung,CN=PsKurs,CN=Local" }
```

Wem das am Anfang noch etwas zu speziell ist, holt alle Benutzerkonten und filtert die gewünschten Konten per *Where-Object*-Cmdlet.

Beispiel

Der folgende Befehl gibt alle Benutzerkonten aus, in deren DistinguishedName das Wort „Verwaltung“ enthalten ist.

```
Get-AdUser -Filter * | Where-Object DistinguishedName -like
"*Verwaltung*"
```

12.3.4 Eingrenzen der Suche

Sobald man die ersten Erfolgserlebnisse hinter sich gelassen hat, möchte man die Suche verfeinern und zum Beispiel nicht bei jedem Aufruf das gesamte Verzeichnis durchsuchen oder mehrere Tausend Ergebnisse erhalten. Die Arbeitsweise der Suche kann über verschiedene Parameter von *Get-ADUser* eingeschränkt werden:

>*SearchBase*
>*SearchScope*
>*ResultSetSize*
>*ResultPageSize*

12.3.5 Suchen nach anderen AD-Objekten

Ein Active Directory-Verzeichnis enthält nicht nur Konten und Gruppen, sondern kann beliebige Sorten von Gegenständen enthalten, zum Beispiel Drucker. Alle diese Gegenstände werden über das *Get-AdObject*-Cmdlet gefunden.

Beispiel

Der folgende Befehl sucht nach dem Standort eines bestimmten Druckers.

```
Get-ADObject -Filter "Name -like '*Brother*'" -Properties Location
```

Voraussetzung ist natürlich immer, dass diese Gegenstände auch in das Verzeichnis eingepflegt wurden, zum Beispiel über das *New-ADObject*-Cmdlet. Die Sorte des anzulegenden Objekts wird über den *Type*-Parameter festgelegt, dem der Name einer Schemaklasse übergeben wird.

Beispiel

Der folgende Befehl legt einen Kontakt im Verzeichnis an.

```
New-ADOBject Name "Peter Monadjemi" -Type Contact
```

Beispiel

Der folgende Befehl gruppiert alle Objekte im AD nach ihrem Typ und sortiert die Ausgabe nach der Häufigkeit der einzelnen Typen.

```
Get-ADObject -Filter * | Group-Object –Property ObjectClass | Sort-Object
–Property Count –Desc
```

▶ **Tipp** Die zur Auswahl stehenden Schemata lassen sich per ADSIEdit betrachten, indem beim Herstellen der Verbindung als Namenskontext „Schema“ ausgewählt wird.

12.4 Benutzer anlegen per New-AdUser

Über das *New-AdUser*-Cmdlet wird ein Benutzerkonto angelegt. Im einfachsten Fall wird lediglich der Name benötigt.

Beispiel

Der folgende Befehl legt ein neues Benutzerkonto an.

```
New-ADUser –Name TestKonto
```

Das Benutzerkonto ist noch nicht aktiv und es wurde noch kein Kennwort vergeben.

▶ **Tipp** Mehrere Parameterwerte lassen sich dank „Parameter-Splatting“ in Gestalt einer Variablen übergeben, die auf einer Hashtable basiert. Damit lässt sich ein Satz von Parameterwerten festlegen, der mehrfach übergeben werden kann. Das spart Tipparbeit und vermeidet Fehler. Die Hashtable-Variable wird mit @ und nicht mit $ angegeben.

12.4.1 Benutzerkonten aktivieren

Wird ein Benutzerkonto nicht bereits beim Anlegen per *New-ADUser* aktiviert, kann dies per *Enable-AdAccount*-Cmdlet jederzeit nachgeholt werden. Dazu benötigt es ein Kennwort, das zuvor per *Set-AdAccountPassword*-Cmdlet gesetzt werden kann. Wird bei diesem Cmdlet sein *PassThru*-Parameter verwendet, wird das Benutzerkonto als *ADUser*-Objekt damit an das *Enable-AdAccount*-Cmdlet weitergereicht.

Beispiel

Der folgende Befehl holt ein Benutzerkonto, das danach ein Kennwort erhält und aktiviert wird.

```
$PwSec = "demo+123" | ConvertTo-SecureString –AsPlainText –Force
Get-ADuser -Identity PsUser | Set-ADAccountPassword -NewPassword $PwSec -
PassThru | Enable-ADAccount
```

12.4.2 Benutzerkonten über eine CSV-Datei anlegen

Sind die Daten für die Benutzerkonten in einer CSV-Datei enthalten, wird diese per *Import-CSV* geladen und per Pipe an *New-AdUser* übergeben. Damit es funktioniert, müssen die Spaltennamen den Namen der Parameter entsprechen. Der *SamAccountName* muss in dieser Varianten explizit gesetzt werden. Ansonsten kann jedes Benutzerkonto im Rahmen von *ForEach-Object* auch einzeln per *New-AdUser* angelegt werden.

Beispiel

Das folgende Beispiel verwendet die Zeilen einer CSV-Datei, um mit den Spaltenwerten Benutzerkonten anzulegen.

```
Import-CSV –Path ADUser.csv | ForEach {
 New-ADUser –Name $_.Name –EMail $_.EMail
}
```

12.5 Benutzerkonten ändern per Set-ADUser

Einzelne Attribute eines Benutzerkontos werden per *Set-ADUser*-Cmdlet geändert. Das Benutzerkonto wird entweder per *Get-AdUser* geholt und über die Pipeline übergeben oder per *Identity*-Parameter ausgewählt.

Beispiel

Der folgende Befehl setzt die E-Mail-Adresse eines Benutzerkontos.

```
Set-AdUser -Identity "CN=User1,OU=PsKurs,DC=pskurs,DC=local" `
 -EMailAddress user1@pskurs.local
```

▶ **Tipp** Die Webseite „SelfADSI“ unter http://selfadsi.de ist für das Herausfinden der Attributnamen immer noch eine wertvolle Hilfe, auch wenn sie seit Jahren nicht mehr aktualisiert wurde. Unter anderem zeigt sie die Namen der LDAP-Attribute in den AD-Dialogfeldern an.

12.5.1 Umgang mit Mehrwert-Attributen

Ein Mehrwert-Attribut (engl. „multi-value attribute“) ist ein Attribut, das aus mehreren Werten besteht. Ein Beispiel von vielen ist das Attribut „OtherHomePhone“, das beliebig viele Telefonnummern enthalten kann. Für den Umgang mit solchen Attributen bietet das *Set-ADUser*-Cmdlet die Parameter *Add*, *Remove*, *Replace* und *Clear*.

Beispiel

Der folgende Befehl fügt zu *OtherHomePhone* eine weitere Telefonnummer hinzu:

```
Set-ADUser -Identity PsUser -Add @{OtherHomePhone=9999}
```

Der folgende Befehl entfernt einen Wert aus dem *OtherHomePhone*-Attribut:

```
Set-ADUser -Identity PsUser -Remove @{OtherHomePhone=9999}
```

Beispiel

Der folgende Befehl ersetzt alle Attributwerte durch neue Werte.

```
Set-ADUser -Identity PsUser -Replace @{OtherHomePhone="1111", "2222",
"3333"}
```

Beispiel

Der folgende Befehl löscht alle Werte eines Attributs.

```
Set-ADUser -Identity PsUser -Clear OtherHomePhone
```

12.6 Benutzerkonten löschen mit Remove-ADUser

Das *Remove-ADUser*-Cmdlet entfernt ein Benutzerkonto. Genau wie beim *Set-ADUser*-Cmdlet werden die zu löschenden Benutzerkonten entweder per Filter geholt oder ein Benutzerkonto wird per Identity-Parameter ausgewählt.

▶ **Hinweis** Wurde der AD-Papierkorb aktiviert (was auch per PowerShell geht), lassen sich gelöschte Benutzerkonten per *Restore-ADUser* problemlos wiederherstellen.

Beispiel

Der folgende Befehl löscht alle Benutzerkonten, in deren Namen ein „Psuser“ enthalten ist.

```
Get-ADUser –Filter { Name –like "PsUser*" } | Remove-ADUser
```

Jedes Benutzerkonto wird nur nach expliziter Bestätigung gelöscht. Über den *Confirm*-Parameter kann die Bestätigungsanforderung generell außer Kraft gesetzt werden.

Beispiel

Der folgende Befehl löscht alle Benutzerkonten, in deren Namen ein „Psuser“ enthalten ist, ohne eine Bestätigung.

```
Get-ADUser –Filter { Name –like "PsUser*" } | Remove-ADUser –
Confirm:$false
```

12.7 Nach Kontenattributen suchen

Geht es bei einer Suche ausschließlich um Attribute, die etwas mit dem Benutzerkonto zu tun haben, ist das *Search-ADAccount*-Cmdlet besser geeignet, da es zahlreiche Parameter anbietet, über die sich nach bestimmten Zuständen bei Benutzerkonten suchen lässt. Ein Beispiel ist die Suche nach abgelaufenen Konten. Die zur Auswahl stehenden Parameter sind *AccountDisabled*, *AccountExpired*, *AccountExpiring*, *AccountInactive*, *Lockedout*, *PasswordExpired* und *PasswordNeverExpires*.

Beispiel

Der folgende Befehl gibt alle abgelaufenen Konten zurück.

```
Search-AdAccount –AccountExpired
```

Beispiel

Der folgende Befehl gibt alle in 14 Tagen ablaufenden Benutzerkonten zurück.

```
Search-ADAccount –AccountExpiring –TimeSpan "14"
```

12.8 Gruppenzugehörigkeiten verwalten

Für die Verwaltung in einem Active Directory-Verzeichnis gibt es mehrere Cmdlets:

- *Add-AdGroupMember*
- *Get-AdGroupMember*
- *New-AdGroup*
- *Get-AdPrincipalGroupMembership*

Beispiel

Der folgende Befehl listet alle Sicherheitsgruppen auf.

```
Get-Adgroup -Filter { GroupCategory -eq "Security" }
```

Beispiel

Der folgende Befehl legt eine neue Sicherheitsgruppe an.

```
New-AdGroup -Name PsKurs -GroupScope DomainLocal -GroupCategory Security
```

Beispiel

Der folgende Befehl fügt drei Benutzerkonten zu einer Gruppe hinzu, die über den *Identity*-Parameter ausgewählt wird. Die Namen der hinzufügenden Benutzerkonten werden über den *Members*-Parameter ausgewählt.

```
Add-ADGroupMember -Identity PsKurs -Members Psuser1, PUser2, PsUser3
```

Gruppenmitgliedschaften werden nicht über das *Memberof*-Attribut, sondern über die Cmdlets *Get-ADGroupMember* und *Get-AdPrincipalGroupMembership* verwaltet. Während *Get-ADGroupMember* die Mitglieder einer bestimmten Gruppe zurückgibt, gibt *Get-AdPrincipalGroupMembership* an, in welchen Gruppen ein bestimmter Benutzer Mitglied ist.

Beispiel

Der folgende Befehl gibt die Mitglieder der Gruppe „PsKurs“ aus.

```
Get-ADGroupMember –Identity PsKurs
```

Da eine Gruppe wiederum Gruppen als Mitglieder beinhalten kann, gibt es bei *Get-ADGroupMember* den *Recursive*-Parameter. Dieser bewirkt, dass auch die Mitglieder dieser Gruppen zurückgegeben werden. Da in diesem Fall ausschließlich Benutzerkonten zurückgegeben werden, erhält man auf sehr einfache Weise sämtliche Benutzerkonten einer Gruppe.

Beispiel

Der folgende Befehl gibt die Namen aller Gruppen aus, in denen das Administrator-Konto Mitglied ist.

```
Get-ADPrincipalGroupMembership -Identity Administrator | Select Name
```

12.8.1 Nicht alles passt zusammen

Nicht alles, was logisch erscheint, funktioniert auch. Ein Beispiel ist das Hinzufügen von Benutzerkonten zu einer Gruppe, die über *Get-ADUser* geholt werden.

Der folgende Befehl funktioniert nicht:

```
Get-AdUser –Filter { City –eq "Esslingen" } | Add-AdGroupMember
"EsslingenGruppe"
```

Der Grund ist, dass der Parameter *Members* seine Werte nicht aus der Pipeline holen kann. Die Umsetzung muss daher Benutzerkonto für Benutzerkonto erledigt werden.

Beispiel

Das folgende Beispiel fügt das Ergebnis einer Abfrage einer Gruppe hinzu.

```
Get-AdUser –Filter { City –eq "Esslingen" } | ForEach {
  Add-AdGroupMember -Identity "EsslingenGruppe" –Members
$_.DistinguishedName
}
```

12.9 Computerkonten abfragen per Get-AdComputer

Das *Get-AdComputer*-Cmdlet holt auf dieselbe Weise Computerkonten wie *Get-ADUser*-Benutzerkonten. Die Konten werden entweder per *Filter*-Parameter gesucht oder per *Identity*-Parameter gezielt ausgewählt.

Beispiel

Der folgende Befehl holt alle Computerkonten in der Domäne.

```
Get-AdComputer –Filter *
```

Beispiel

Die folgende Abfrage gruppiert alle Computerkonten in der Domäne nach ihrer Betriebssystembezeichnung.

```
Get-ADComputer –Filter * -Properties OperatingSystem | Group –Property
OperatingSystem
```

12.10 Abfrageergebnisse mit Out-GridView kombinieren

Durch Kombination von Cmdlets wie *Get-AdUser* oder *Get-ADComputer* mit dem *Out-GridView*-Cmdlet bei gesetztem *PassThru*-Parameter werden die gefundenen Benutzerkonten in einem Fenster aufgelistet. Nach der Auswahl einzelner Einträge und einer Bestätigung wird eine Operation wie zum Beispiel *Remove-AdUser* mit allen ausgewählten Benutzerkonten durchgeführt. Damit ergibt sich eine einfache

Kombination eines PowerShell-Abfragebefehls mit einer komfortablen Auswahlmöglichkeit in einem Fenster.

12.11 Umgang mit Organisationseinheiten

Eine Organisationseinheit (kurz „OU“ für „Organizational Unit“) ist ein logischer Container für andere Objekte. Für das Abfragen, Anlegen, Ändern und Entfernen von Organisationseinheiten (OUs) stehen die Cmdlets *Get-ADOrganizationalUnit*, *New-ADOrganizationalUnit*, *Set-ADOrganizationalUnit* und *Remove-ADOrganizationalUnit* zur Verfügung. Objekte werden per *Move-ADObject* in eine OU verschoben.

Beispiel

Der folgende Befehl holt alle OUs, die ein bestimmtes Kriterium erfüllen, und gibt ihre Eckdaten aus.

```
Get-ADOrganizationalUnit -Filter * | Select Name, Description, ManagedBy
```

Beispiel

Der folgende Befehl legt eine neue OU mit dem Namen „TestOU“ auf der obersten Ebene des Verzeichnisses an.

```
New-ADOrganizationalUnit -Name TestOU -Path "DC=PsKurs, DC=local"
```

Beispiel

Der folgende Befehl verschiebt ein paar Benutzerkonten in die OU „TestOU“.

```
Get-AdUser -filter { Name -like "TestUser*"} | Move-ADObject -TargetPath
"OU=TestOU,DC=mobil,DC=pshub,DC=local"
```

Beispiel

Der folgende Befehl gibt den Inhalt einer OU aus, indem diese bei *Get-ADUser* über den Parameter *SearchBase* ausgewählt wird.

```
Get-ADUser -Filter * -SearchBase "OU=TestOU,DC=mobil,DC=pshub,DC=local"
```

12.12 Das AD-Laufwerk

Mit dem Laden des ActiveDirectory-Moduls wird ein Laufwerk mit dem Namen AD hinzugefügt, das vom Provider „ActiveDirectory“ zur Verfügung gestellt wird. Es erlaubt das Ansprechen des AD-Verzeichnisses als Laufwerk mit Cmdlets wie *Get-ChildItem*,

New-Item, *Remove-Item* usw. Besonders interessant für den Umgang mit AD-Berechtigungen über *Get-ACL* und *Set-ACL*. Als Pfad muss in der Regel der DN angegeben werden.

Beispiel

Der folgende Befehl gibt den Inhalt des User-Containers aus.

```
dir "ad:\Cn=Users,Dc=pskurs,Dc=Local"
```

Beispiel

Der folgende Befehl führt über den *Filter*-Parameter von *Get-ChildItem* eine Suche durch.

```
dir "ad:\Cn=Users,DC=mobil,DC=pshub,DC=Local" -Filter "CN=*ABC*"
```

Beispiel

Dass über den *Filter*-Parameter allgemeine LDAP-Attribute angegeben werden können, macht das nächste Beispiel deutlich.

```
dir "ad:\Cn=Users,Dc=mobil,Dc=pshub,Dc=Local" -Filter "l=Esslingen"
```

Der Provider unterstützt das Anlegen neuer PSDrives.

Beispiel

Das folgende Beispiel legt ein Laufwerk „ITOU“ an, über das eine OU etwas kürzer angesprochen werden kann.

```
New-PSDrive –Name ITOU –Root "AD:\OU=IT_OU,DC=pskurs,DC=local" –
PSProvider ActiveDirectory
```

12.13 Den AD-Papierkorb aktivieren

Bei seiner Einführung mit Windows Server 2008 R2 konnte der AD-Papierkorb nur per PowerShell und dem Enable-*ADOptionalFeature*-Cmdlet aktiviert werden. Seit Windows Server 2012 ist dies auch im Rahmen des AD-Verwaltungscenters möglich.

Beispiel

Die folgende Befehlsfolge aktiviert den AD-Papierkorb für die angegebene Domäne.

```
$DomName = "pskurs.local"
$DomDN = "DC=pskurs,DC=local"
Enable-ADOptionalFeature `
  -Identity "CN=Recycle Bin Feature, CN=Optional Features, CN=Directory
    Service, CN=Windows NT, CN=Services, CN=Configuration, $DomDN" `
-Scope ForestOrConfigurationSet -Target $DomName
```

12.14 Einen AD LSD oder Open LDAP-Server ansprechen

Das Abfragen eines AD LSD-Servers (*Active Directory Lightweight Services*) oder eines Open LDAP-Servers ist mit den Hausmitteln der PowerShell relativ einfach möglich. Im Mittelpunkt stehen die Typenaliase *[ADSI]* und *[ADSISearcher]*. *[ADSI]* steht für den Typ *System.DirectoryServices.DirectoryEntry*, der einen Eintrag in einem Verzeichnis repräsentiert. *[ADSISearcher]* steht für den Typ *System.DirectoryServices.DirectorySearcher*, über den eine Suche in einem LDAP-Verzeichnis durchgeführt wird.

Beispiel

Das folgende Beispiel führt eine einfache Abfrage gegen einen öffentlichen LDAP-Server durch. Die Rückgabe sind die Namen berühmter Wissenschaftler.

```
<#
 .Synopsis
 Zugriff auf einen Open LDAP-Server
#>

$Server = "LDAP://ldap.forumsys.com:389/DC=example,DC=com"
$DN = "cn=read-only-admin,dc=example,dc=com"

$Filter = "(objectClass=*)"
$Pw = "password"

$DirEntry = New-Object -TypeName System.DirectoryServices.DirectoryEntry
-ArgumentList $Server, $DN, $Pw, "FastBind"
$DirSearcher = New-Object -TypeName
System.DirectoryServices.DirectorySearcher -ArgumentList $DirEntry,
$Filter
$DirSearcher.FindAll() | Select -ExpandProperty Properties | ForEach {
    New-Object -TypeName PsObject -Property @{
                                              CN = $_["cn"]
                                              Class=$_["Objectclass"]
                                              Path = $_["adspath"]
                                              }
}
```

Der „Trick“, der offiziell natürlich keiner ist, besteht darin, als weiteres Argument beim Anlegen des *DirectoryEntry*-Objekts den Wert „FastBind“ zu übergeben.

Per *[ADSI]* und *[ADSISearcher]* kann auch ein „reguläres“ AD abgefragt werden. Im Vergleich zum *ActiveDirectory*-Modul ist diese Variante aber deutlich umständlicher. Sie ist nur dann eine Option, wenn das Modul aus irgendeinem Grund nicht zur Verfügung stehen sollte. Die beiden Typenalias besitzen aber trotzdem ihre Berechtigung. Sie werden immer dann benötigt, wenn ein LDAP-Verzeichnis angesprochen werden soll, das auf den *Active Directory Leightweight Services* (AD LDS) basiert, die zum Beispiel auch unter Windows 7 installiert werden können. AD LDS stellt einen LDAP-Verzeichnisdienst zur Verfügung, allerdings ohne Anmeldeauthentifizierung und die Notwendigkeit für einen Domänenkontroller. AD LDS wird bei Windows 7 beziehungsweise Windows 10 über die Programme und Features hinzugefügt.

Beispiel

Der folgende Befehl führt per *[ADSISearcher]* eine Suche in einem AD LDS-Verzeichnis aus, die alle Objekte vom Typ „person“ zurückgibt, deren Name mit „P“ beginnt.

```
$ADS = [ADSISearcher]"(&(ObjectClass=person)(name=P*))"
ADS.SearchRoot= "LDAP://localhost:389/CN=documents"
$ADS.FindAll() | ForEach {
    [PSCustomObject]@{Name=$_.Properties["name"][0]
                      Beschreibung=$_.Properties["description"][0]
                     }
}
```

Nicht nur, dass die Rückgabe aus einer Collection besteht, auch die LDAP-Filtersyntax besitzt so ihre Besonderheiten. Dazu gehört zum Beispiel der Umstand, dass Vergleichsausdrücke in „Umgekehrt Polnischer Notation“ (UPN) angegeben werden und es bei den Attributnamen auf die Groß-/Kleinschreibung ankommt.

12.15 Zusammenfassung

Das ActiveDirectory-Modul umfasst als Teil von Windows Server (je nach Version) über 140 Cmdlets. Es steht ab Windows Server 2008 R2 als Feature zur Verfügung und ist für Windows 7/8.1 und Windows 10 Teil der RSAT. Mit Hilfe der Cmdlets wird die Administration eines Active Directory per Befehlszeile und Skript sehr einfach.

Azure-Administration per PowerShell

13

Zusammenfassung

In diesem Kapitel geht es um das Thema Cloud. Konkret geht es um die Microsoft-Cloud-Plattform Azure, die seit ihrer Einführung stetig wächst und auf der inzwischen über 100 verschiedene Dienste angeboten werden, Tendenz weiter steigend (eine vollständige Übersicht gibt es unter https://azure.microsoft.com/de-de/services). Noch konkreter geht es in diesem Kapitel darum, wie diese Dienste per PowerShell angesprochen werden können. Die Beispiele in dem Kapitel basieren auf dem Azure Resource Manager (ARM). Das alte, auf einem Service Manager basierende Modell, und damit die Azure-Cmdlets der ersten Generation, werden in diesem Kapitel nicht behandelt.

13.1 Ein erster Überblick

Azure ist kein neues Thema. Die Microsoft Cloud-Plattform gibt es offiziell bereits seit 2010, entsprechend lange gibt es auch PowerShell-Cmdlets, die allgemein unter dem Begriff „Azure PowerShell“ zusammengefasst werden. Ein wichtiger Einschnitt in der noch jungen Geschichte der Plattform war die Einführung des *Azure Ressourcen Managers* (ARM) im Jahr 2014. Anstatt beim Anlegen einer Azure-Funktionalität jede Einstellung einzeln vornehmen zu müssen, geschieht dies beim ARM auf der Grundlage von Vorlagen (Templates). Ressourcen wie Speicher werden nicht direkt, sondern indirekt über Ressourcengruppen zugeordnet – eine naheliegende Idee, die von einem bekannten Mitbewerber vor der Einführung in Gestalt der Cloudformations bereits verwendet wurde. Durch die Umstellung auf den ARM entstand ein komplett neuer Satz an Azure-Cmdlets als Teil der Azure PowerShell. Wie umfangreich die Azure-Funktionalität inzwischen geworden ist, macht der Umstand deutlich, dass die Azure PowerShell aktuell (Stand: April 2017) aus 37 Modulen mit 1258 Cmdlets besteht.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_13

Wer sich im Internet nach Know-how umschaut, findet entsprechend viele Beispiele. Ein Umstand, der am Anfang für etwas Konfusion sorgen könnte ist, dass sich viele Beispiele noch auf die alte Azure PowerShell beziehen. Microsoft hat die Umstellung des Azure-Portals von „Classic“ auf das neue Layout, bei dem der ARM im Mittelpunkt steht, noch nicht vollständig abgeschlossen, so dass es nach wie vor möglich ist, Azure-Funktionalitäten wie virtuelle Maschinen anzulegen, die auf dem alten Modell basieren.

Die ARM-Cmdlets sind daran zu erkennen, dass sie ein „Rm“ im Noun-Anteil des Namens tragen. Ein Beispiel ist das *Select-AzureRmProfile*-Cmdlet aus dem *AzureRm.Profile*-Modul. Ein Cmdlet wie Get-AzureVM stammt aus dem alten Azure-Modul. Dieses Cmdlets können natürlich weiterhin verwendet werden, sie sollten aber nicht mehr verwendet werden. Dem ARM-Modell gehört bei Azure die Zukunft. Das alte Portal und das auf dem Service Manager basierende Model werden wahrscheinlich irgendwann in naher Zukunft offiziell abgekündigt werden. Spätestens dann müssten PowerShell-Skripte auf den ARM umgestellt werden. Dabei geht es nicht nur um unterschiedliche Namen, auch die Herangehensweise ist anders als beim eventuell vertrauten Service Manager-Modell.

13.2 Der ARM im Überblick

Die Rolle des Azure Ressource Managers (ARM) lässt sich einfach beschreiben: Mit seiner Hilfe werden beliebige Ressourcen wie öffentliche IP-Adressen, Netzwerkadapter, Endpunkte, Speicherkonten oder VMs zu einer logischen Einheit zusammengefasst. Diese logische Einheit heißt Ressourcengruppe („Resource Groups“). Ressourcengruppen sind immer einem Abonnement (Subscription) zugeordnet. Durch Ressourcengruppen wird das Anlegen einer Azure-Funktionalität deutlich vereinfacht. Ein Beispiel ist das Anlegen einer Website. Alle „Zutaten“ wie Datenbank, Anwendungsanalyse via Application Insight und die Dateien der Website selber werden zu einer Ressourcengruppe zusammengefasst, die gemeinsam administriert wird.

Technisch ist der ARM eine Schnittstelle, die die Funktionalität der Azure-Plattform über eine REST-API zugänglich macht. Über eine rollenbasierte Zugriffskontrolle (RBAC für „Role Based Access Control“) kann der Zugriff auf einzelne Ressourcen eingeschränkt werden, so dass zum Beispiel das Bereitstellen von Ressourcen nur bestimmten Benutzergruppen möglich ist. Abb. 13.1 fasst die Bestandteile des ARM in einem Schaubild zusammen.

13.2.1 Die Rolle der Resource Provider

Die einzelnen Ressourcen werden über Resource Provider zur Verfügung gestellt, die jeweils für eine bestimmte Sorte von Ressourcen zuständig sind. Ein Beispiel ist der Compute-Provider, der unter anderem virtuelle Maschinen zur Verfügung stellt. Insgesamt gibt

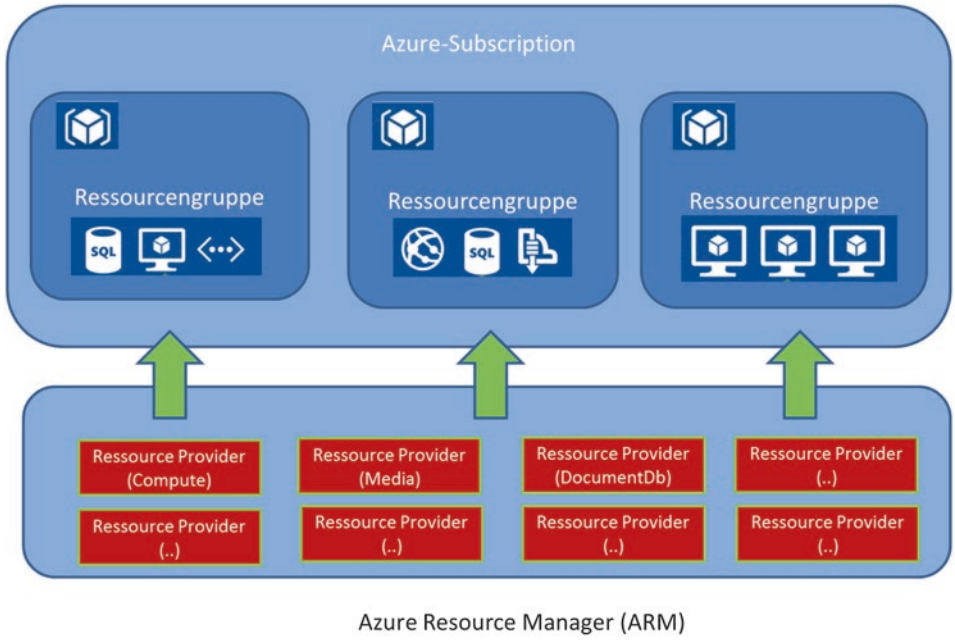

Abb. 13.1 Der Azure Resource Manager im Überblick

es mehrere Dutzend dieser Provider, die sich auch per PowerShell und das Cmdlet *Get-AzureRmResourceProvider* aus dem *AzureRm*-Modul auflisten lassen.

Ressourcengruppen werden im Azure-Portal angelegt (Abb. 13.2). Für das Anlegen einer Gruppe werden lediglich ein Name und eine Region benötigt.

13.2.2 Die Rolle der Templates

Die stärkste Eigenschaft des ARM ist, dass Ressourcen auf der Basis von Templates angelegt werden können. Ein Template ist eine Vorlage für das Anlegen einer bestimmten Azure-Funktionalität, wie zum Beispiel einer virtuellen Maschine. Das Template-Konzept ist sehr flexibel. Es lassen sich beliebige Kombinationen von Azure-Ressourcen anlegen und mit Parametern parametrisieren. Ein Template basiert auf einer Textdatei, die eine textuelle Beschreibung der Ressourcen im JSON-Format enthält. Die Beschreibungsnotation kennt, vergleichbar mit einer Skriptsprache, unter anderem Datentypen, Operatoren und einfache Funktionen, so dass sich Templates flexibel parametrisieren lassen. Bemerkenswert ist, dass sich über Elemente wie „if“ und „then“ auch Entscheidungen abbilden lassen, die bei der Auswertung des Templates ausgeführt werden.

Abb. 13.2 Für das Anlegen einer Ressourcengruppe wird lediglich ein Name benötigt

Dass ein Template nicht kompliziert sein muss, macht der folgende Rahmen deutlich, der das erforderliche Minimum an Deklarationen enthält:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
    },
    "variables": {
    },
    "resources": [
    ],
    "outputs": {

    }
}
```

Dass ein Template in der Praxis deutlich komplexer ist, machen die zahlreichen Beispiele deutlich, die Microsoft über das GitHub-Portal zur Verfügung stellt. Templates werden nicht mit einem Texteditor erstellt, da in dieser Umgebung keine Eingabehilfen zur Verfügung stehen. Neben Microsoft Visual Studio kommt für das Erstellen von Templates auch das deutlich schlankere Visual Studio Code in Frage, in dem ebenfalls dank Schemaunterstützung Auswahllisten und andere Eingabehilfen angeboten werden.

Beispiel

Der folgende Ausschnitt aus einem JSON-Template definiert die Eckdaten für eine VM:

```
{
    "apiVersion": "2015-06-15",
    "type": "Microsoft.Compute/virtualMachines",
      "name": "[variables('vmName')]",
     "location": "[resourceGroup().location]",
      "tags": {
     "displayName": "PemoMachine"
    },
    "dependsOn": [
      "[concat('Microsoft.Storage/storageAccounts/',
variables('vhdStorageName'))]",
      "[concat('Microsoft.Network/networkInterfaces/',
variables('nicName'))]"
    ],
    "properties": {
      "hardwareProfile": {
          "vmSize": "[variables('vmSize')]"
        },
}
```

Um das Template flexibel nutzen zu können, kommen Variablen zum Einsatz. Eine solche ist zum Beispiel „vmName“, die an einer anderen Stelle der Template-Datei einen Wert erhält. Templates besitzen Parameter. Diese werden beim Bereitstellen des Templates entweder direkt oder indirekt über eine weitere Datei, die Parameterdatei, zur Verfügung gestellt.

Ein Template muss nicht selber erstellt werden, im zuständigen GitHub-Portal steht eine reichhaltige Auswahl einsatzfertiger QuickStart-Templates zur Verfügung (https://github.com/Azure/azure-quickstart-templates). Die Auswahl wird unter Mitwirkung der Community stetig erweitert. Mit den Templates hält auch bei Azure der deklarative Ansatz Einzug, vergleichbar mit DSC (*Desired State Configuration*). Anstatt das Anlegen von Azure-Ressourcen Schritt für Schritt für Befehle festzulegen, wird der gewünschte Zustand durch eine Textdatei beschrieben, die am Ende nur angewendet werden muss.

Wem das GitHub-Portal für den direkten Download zu technisch ist, dem bietet Microsoft ein eigenes Portal als Alternative an:

https://azure.microsoft.com/en-us/documentation/templates/

Hier besteht die Möglichkeit, nach Templates anhand von Stichworten zu suchen (Abb. 13.3). Ein gefundenes Template wird über den **Deploy to Azure**-Button direkt im Azure-Portal bereitgestellt und kann damit sofort angewendet werden (Abb. 13.4).

Eine faszinierende Alternative ist die Option, ein Template im Azure Resource Visualizer zu visualisieren. Dahinter steckt eine kleine Webanwendung mit der Adresse http://armviz.io. Der Inhalt der JSON-Datei wird dabei in einem Designer im Azure-Portal dargestellt (Abb. 13.5). Das Template kann an dieser Stelle bearbeitet und mit Ressourcenelementen aus einer Toolbox ergänzt werden. Es ist davon auszugeben, dass diese Funktionalität in Zukunft auch direkt im Azure-Portal zur Verfügung stehen wird.

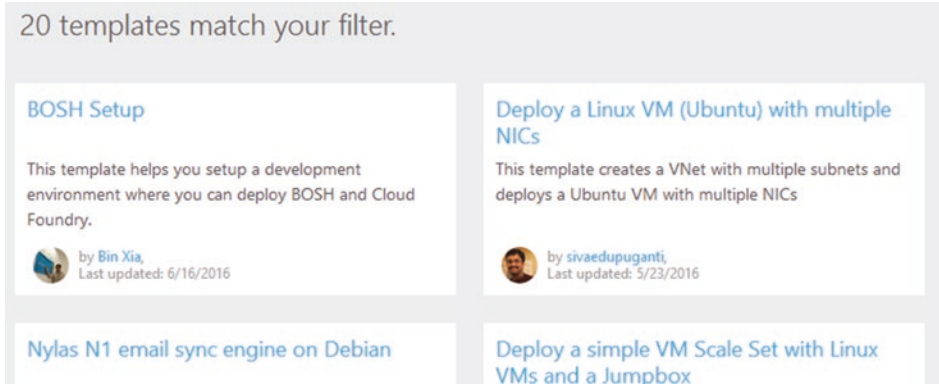

Abb. 13.3 Microsoft bietet ARM-Templates über ein Webportal zum Download

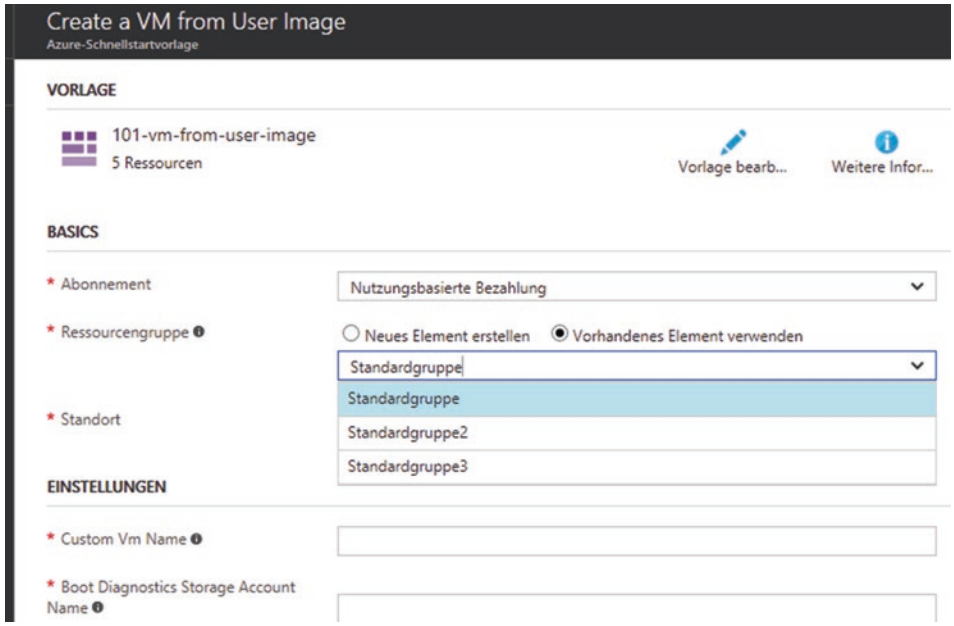

Abb. 13.4 Das unter GitHub ausgewählte Template steht kurz danach im Azure-Portal zur Verfügung

Wer ein Template doch „zu Fuß“ erstellen möchte, verwendet dazu entweder einen Editor, Visual Studio Code, das nach der Installation einer Extension Eingabehilfen anbietet, oder Visual Studio (zum Beispiel in der kostenlosen Community-Edition), das Vorlagen für Azure-Templates anbietet, wenn zuvor das Azure SDK installiert wurde (Abb. 13.6). Ab der Version 2.9 können die einzelnen Komponenten auch einzeln heruntergeladen

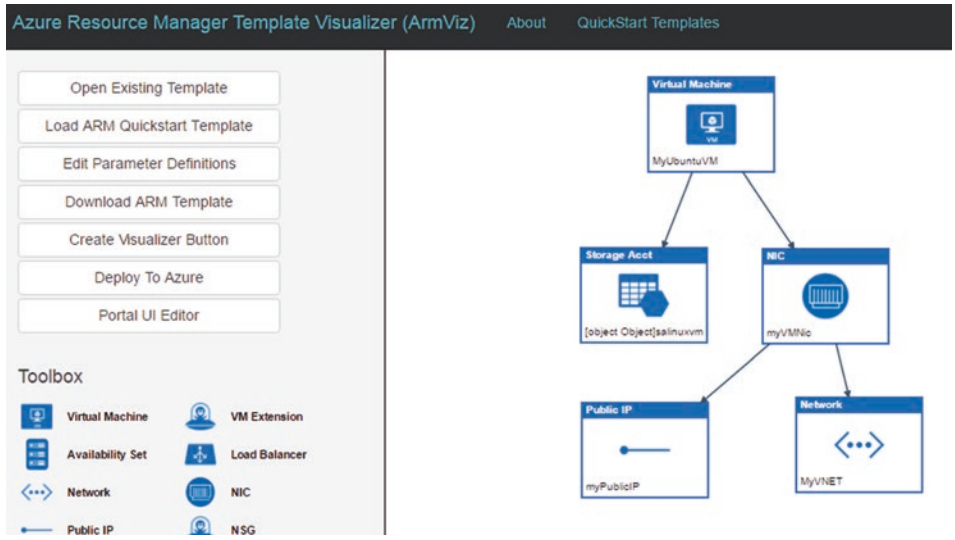

Abb. 13.5 Ein Template wird im Azure-Designer visuell dargestellt

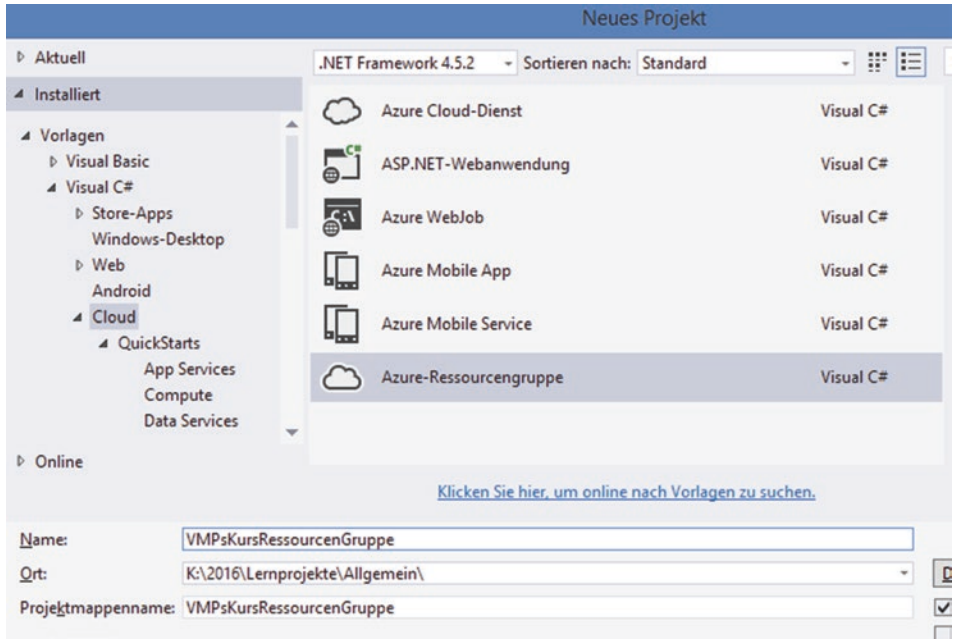

Abb. 13.6 Dank der Azure Tools aus dem Azure SDK steht in Visual Studio eine Vorlage für eine Ressourcengruppe zur Auswahl

werden. Nach der Installation des Azure SDKs enthält die Projektkategorie **Azure** in Visual Studio die Vorlage „Azure-Ressourcengruppe“. Nach der Auswahl der Vorlage muss lediglich der geplante Ressourcentyp ebenfalls über eine Vorlage ausgewählt werden (Abb. 13.7).

Das Ergebnis ist ein Projekt, das neben der JSON-Datei auch ein PowerShell-Skript für die spätere Bereitstellung der Ressource im Azure-Portal enthält.

Die Unterstützung für das etwas sperrige (aber effektive) JSON-Format wird mit jeder Version des Azure SDKs verbessert. Neben Auswahllisten und Fehlerüberprüfungen während der Eingabe kann die Struktur der Vorlage im JSON-Outline-Fenster auch in einer Baumstruktur betrachtet werden (Abb. 13.8). Die Grundlage dafür ist ein Schema (JSON-Schema), das Microsoft entwickelt hat, um den Umgang mit JSON-Daten zu vereinfachen.

Am Ende muss das Projekt lediglich über das beim Anlegen des Projekts automatisch angelegte PowerShell-Skript bereitgestellt werden, damit die Ressource(n) im Azure-Portal unter dem angegebenen Benutzerkonto angelegt werden (Abb. 13.9). Wenn die Vorlage Parameter verwendet, können diese in einem dafür vorgesehenen Dialogfenster editiert werden.

Es erscheint eine Dialogbox für die Eingabe der Variablenwerte, die im Template verwendet werden (Abb. 13.10). Der weitere Verlauf kann im Ausgabefenster von Visual Studio nachvollzogen werden. Dies ist insbesondere dann sehr wichtig, wenn ein Fehler aufgetreten ist.

▶ **Hinweis** Die Vorgehensweise zur Bereitstellung einer Azure-Ressource mit Hilfe von Visual Studio und dem Vorlageneditor wird in der Azure-Dokumentation exemplarisch ausführlich und gut nachvollziehbar beschrieben: https://azure.microsoft.com/de-de/documentation/articles/vs-azure-tools-resource-groups-deployment-projects-create-deploy.

Eine weitere Alternative, die ausschließlich für Entwickler in Frage kommt, sind die Microsoft Azure Management Libraries, die ebenfalls Teil des Azure SDK sind. Dabei

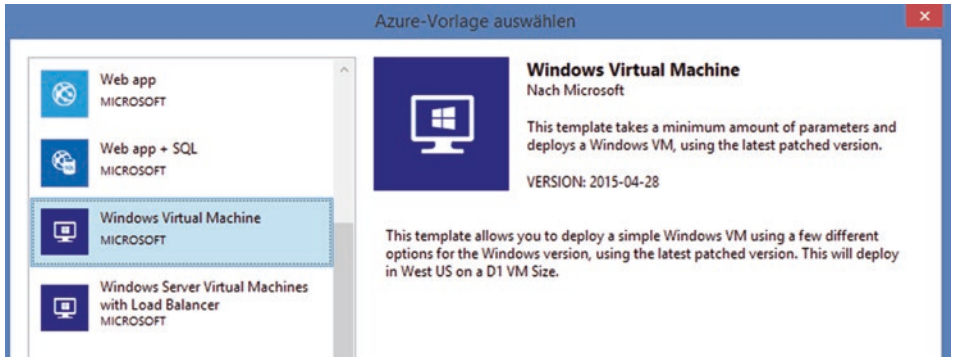

Abb. 13.7 Der neuen Ressourcengruppe wird ein Templatetyp zugeordnet

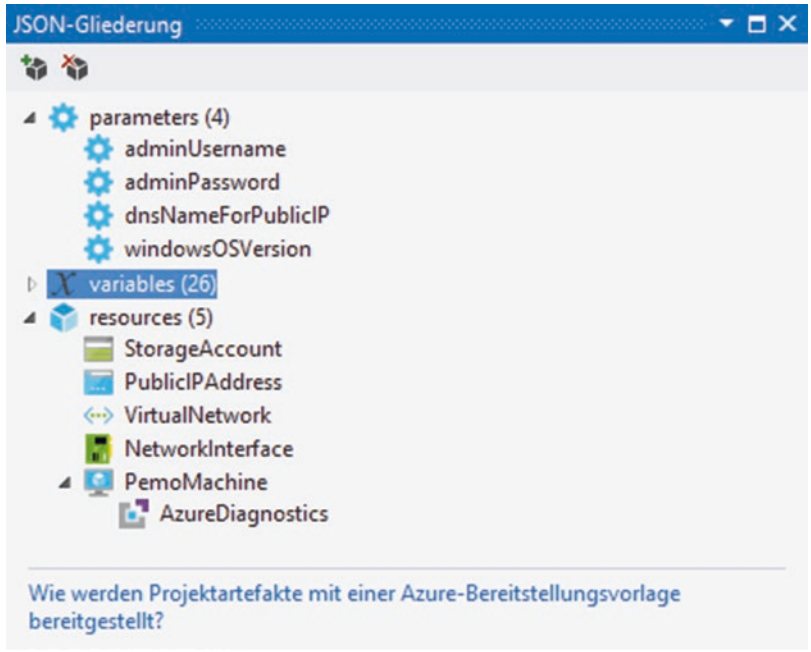

Abb. 13.8 Die JSON-Gliederung zeigt den Inhalt eines Templates übersichtlich an

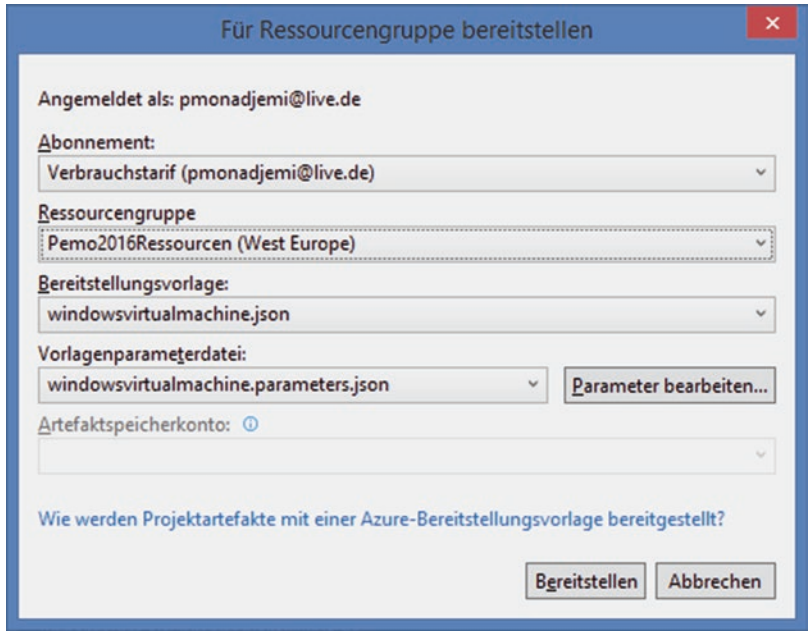

Abb. 13.9 Am Ende wird die Ressourcengruppe unter Azure bereitgestellt

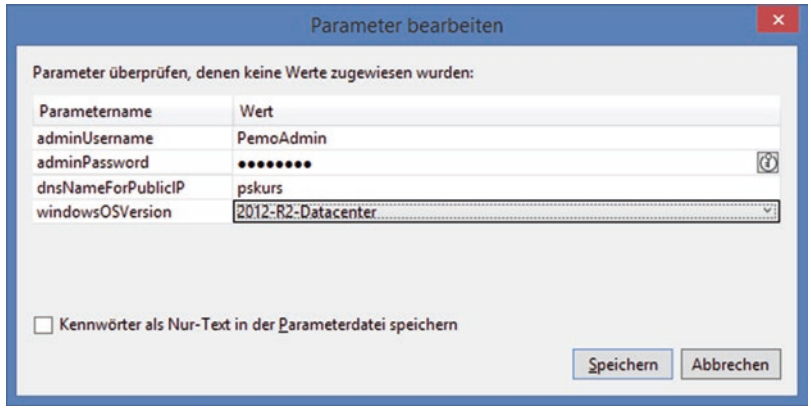

Abb. 13.10 Vor der Bereitstellung werden die Werte für die im Template definierten Variablen abgefragt

handelt es sich um einen Satz von Assemblys, mit deren Hilfe das Anlegen von Azure-Ressourcen über Klassen durchgeführt wird. Ein Template ist hier nicht im Spiel. Nach erfolgreicher Installation der Azure-Tools stehen in der Kategorie „Cloud\QuickStarts“ zahlreiche Beispielprojekte zur Auswahl.

13.3 Zugriffssteuerung per BPAC

Die Umstellung auf den ARM bedeutet auch die Umstellung auf ein rollenbasiertes Zugriffsmodell (BPAC für „Role Based Security Control“). Für jede Ressource existiert eine Zugriffsberechtigung, die auf Rollen basiert. Damit lässt sich der Zugriff auf eine Ressource für bestimmte Benutzergruppen (Rollen) im Detail festlegen. Definiert werden die Sicherheitsgruppen und die dazugehörigen User in einem Azure Active Directory-Tenant (zu Deutsch „Mieter“), der zuvor im Azure-Portal angelegt wurde. Dieser ist mit einem Domänennamen verknüpft, der in der Regel auf „onmicrosoft.com“ endet. Als nächstes wird, entweder im Azure-Portal oder per PowerShell, eine Rollendefinition angelegt, die festlegt, wie einzelne Gruppen einzelne Ressourcen benutzen dürfen. Im nächsten Schritt wird die Rollendefinition einer Sicherheitsgruppe zugewiesen, die dann zum Beispiel auf eine Ressourcengruppe, eine einzelne Ressource oder auf die gesamte Subscription angewendet wird.

Ergänzend stehen optionale ARM-Richtlinien zur Verfügung, über die sich zum Beispiel Namensrichtlinien beim Anlegen von Ressourcen erzwingen oder Ressourcen automatisch einer Region zuordnen lassen.

Eine weitere Sicherheitsmaßnahme sind Resource Locks, über die sich einzelne Ressourcen vor einem unbeabsichtigten Löschen schützen lassen. Locks werden über PowerShell-Cmdlets wie zum Beispiel *New-AzureRmResourceLock* angelegt, können aber auch Teil eines Templates sein.

13.4 Die Azure PowerShell im Überblick

Alle Azure-Cmdlets werden unter den Namen „Azure PowerShell“ zusammengefasst. Sie werden als Teil des Azure SDKS für .NET am einfachsten über den Web Platform Installer (WebPI) installiert. Alternativ kommt inzwischen auch die Installation von der PowerShell Gallery über das *Install-Module*-Cmdlet mit „AzureRm“ als Modulname in Frage.

▶ **Hinweis** Unmittelbar nach der Einführung des ARM kam jemand bei Microsoft auf die Idee, das alle Azure PowerShell-Cmdlets in den beiden Modi ASM und ARM betrieben werden sollten. In der ersten Version der Azure PowerShell wurde das auch tatsächlich umgesetzt. Über das *Switch-AzureMode-Cmdlet* war ein Umschalten zwischen beiden Modi möglich. Nach berechtigter Kritik der Anwender wurde diese Hilfskonstruktion wieder aus dem Verkehr gezogen und die neuen ARM-basierten Cmdlets in eigenen Modulen untergebracht.

13.4.1 Abfragen der PowerShell-Version

Die Azure PowerShell wird aktuell noch offiziell über den Webplatform Installer (WebPI) installiert. Da dieser auf einer Assembly basiert, lassen sich mit ihrer Hilfe die verfügbaren und bereits installierten Versionen der Azure PowerShell abfragen. In Zukunft wird die PowerShell Gallery die offizielle Bezugsquelle sein.

Beispiel

Die Function *Get-WindowsAzurePowerShellVersion* gibt sowohl die verfügbaren Versionen als auch die bereits installierte Version der Azure PowerShell aus. Damit können Sie feststellen, ob Sie mit der neuesten Version arbeiten.

```
function Get-WindowsAzurePowerShellVersion
{
  [CmdletBinding()]
  param()
  Write-Host 'Azure PowerShell installierte Version: ' -ForegroundColor
'Yellow'
  Get-Module -ListAvailable | Where-Object Name -eq 'Azure' | Select
Version, Name, Author
  Write-Host "`nAzure PowerShell verfügbare Version: " -ForegroundColor
'Green'
  $AssName = "Microsoft.Web.PlatformInstaller, Version=5.0.0.0,
Culture=neutral, PublicKeyTo-ken=31bf3856ad364e35"
Add-Type -AssemblyName $AssName
$ProductManager = New-Object -TypeName Mi-
crosoft.Web.PlatformInstaller.ProductManager
$ProductManager.Load()
$ProductManager.Products | Where { $_.Title -match 'Azure PowerShell' -
and $_.Author -eq 'Microsoft Corporation' }  | Select-Object Title,
Version, Published
}
```

13.4.2 Ein erster Überblick

Die „neue“ Azure PowerShell, konkret das AzureRm-Modul, umfasst 42 Module und über 1.589 Cmdlets. Die hohe Anzahl erklärt sich durch den Umstand, dass es für jeden der ARM-Provider ein eigenes Modul gibt. Die Cmdlets für den Compute-Provider, der unter anderem virtuelle Maschinen bereitstellt, befinden sich entsprechend in dem *AzureRM.Compute*-Modul. Ein

```
Get-Command -Module AzureRM.Compute
```

gibt daher aller Cmdlets aus, die etwas mit dem Bereitstellen von virtuellen Maschinen über den ARM zu tun haben.

Da die einzelnen Module, die unter dem AzureRm-Modul zusammengefasst werden, unterschiedlich alt sind, liegen sie auch in unterschiedlichen Versionsnummern vor. Die Cmdlets für den Umgang mit dem ARM selber sind im *AzureRM.Resources*-Modul enthalten. Mit 55 Cmdlets ist das Modul regelrecht überschaubar. Tab. 13.1 stellt die wichtigsten Cmdlets des Moduls mit einer kurzen Beschreibung zusammen.

13.4.3 Die ersten Schritte mit der Azure PowerShell

Die ersten Schritte mit der Azure PowerShell bestehen aus einer Anmeldung am Azure-Portal. Auch hier kann es kurzzeitig zu etwas Verwirrung kommen, denn es kommt darauf an, ob sie sich mit einem Geschäftskonto, das zum Beispiel über Office 365 angelegt wurde, oder mit einem Microsoft-Konto, also mit einer privaten E-Mail-Adresse, anmelden. Mit einem Microsoft-Konto kann (aktuell) der Anmeldedialog nicht umgangen werden. Skripte, die ohne interaktive Anmeldung ausführen sollen, erfordern daher ein Geschäftskonto. Für die Anmeldung selber stehen neben der klassischen Kombination aus Benutzername und Kennwort auch Zertifikate und die Anmeldung mit dem Namen eines Service Principals zur Verfügung. In den Beispielen in diesem Kapitel wird die klassische Form der Anmeldung mit einem *PSCredential*-Objekt verwendet.

- **Hinweis** Die für das alte Azure-Portal verwendete Publishsettings-Datei spielt beim ARM keine Rolle mehr.

- **Hinweis** Die AzureRm-Cmdlets sind „work in progress“. Bei vielen Cmdlets erscheinen bei der Ausführung daher Warnungen, die darauf hinweisen, dass sich in Zukunft etwas ändern wird, zum Beispiel das Ausgabeformat.

Beispiel

Der folgende Befehl führt eine Anmeldung mit der Angabe des Namens der Subscription durch:

```
Login-AzureRmAccount -SubscriptionName "Spezialtarif"
```

Tab. 13.1 Die wichtigsten Cmdlets aus dem Modul AzureRM.Resources

Cmdlet	Was macht es?
Find-AzureRmResource	Holt alle Ressourcen eines bestimmten Typs und mit einem bestimmten Namen.
Find-AzureRmResource-Group	Holt entweder alle Ressourcengruppen oder jene, die ein bestimmtes Tag besitzen.
Get-AzureRmADGroup	Holt alle oder bestimmte AD-Gruppen, die über einen Suchstring ausgewählt wurden.
Get-AzureRmADG-roupMember	Holt die Mitglieder einer AD-Gruppe. Da die Gruppe nur über ihre GUID festgelegt werden kann, wird das Cmdlet in der Regel pipeline-basierend eingesetzt.
Get-AzureRmADUser	Holt alle AD-User oder jene, deren Name einem Suchwort entspricht.
Get-AzureRmPolicy-Definition	Holt alle Richtliniendefinitionen. Von Anfang an sind Richtlinien vorhanden, die das Anlegen von Ressourcen in bestimmten Regionen regeln.
Get-AzureRmResource	Holt alle oder bestimmte Ressourcen anhand ihres Namens und des Namens der Ressourcengruppe. Beim Namen kommt es auf die Groß-/Kleinschreibung an. Gibt es die Ressource nicht, ist ein Fehler die Folge.
Get-AzureRmResource-Group	Holt alle oder bestimmte Ressourcengruppen.
Get-AzureRmResource-GroupDeployment	Holt alle bereits durchgeführten Deployments einer bestimmten Ressourcengruppe.
Get-AzureRmResource-Lock	Holt alle eingerichteten Locks.
Get-AzureRmResource-Provider	Holt die Eckdaten aller registrierten Provider wie zum Beispiel Microsoft.Compute, der unter anderem virtuelle Maschinen zur Verfügung stellt.
Move-AzureRm-Resource	Verschiebt eine Ressource nicht nur in eine andere Ressourcengruppe, sondern ordnet diese auf Wunsch auch einem anderen Abonnement zu. Der Speicherort der Ressource ändert sich nicht. Es lassen sich nicht alle Ressourcentypen verschieben. Das Verschieben virtueller Maschinen ist von ASM (V1) nach ARM (V2) auf diese Weise möglich. Weitere Einschränkungen sind in der Azure-Dokumentation beschrieben.
New-AzureRm-Resource	Legt eine neue Ressource an, die vom ARM verwaltet wird. Eines der flexibelsten Cmdlets aus dem Modul, denn die neue Ressource kann zum Beispiel eine Website oder ein Speicherkonto sein.
New-AzureRmResource-GroupDeployment	Legt ein neues Deployment für eine Ressourcengruppe auf der Grundlage eines Templates an.
Test-AzureRmResource-GroupDeployment	Testet, ob ein Deployment mit einem bestimmten Template für die angegebene Ressourcengruppe funktioniert.

Die Angabe der Subscription ist optional. Es erscheint die übliche Anmeldedialogbox. Die Rückgabe nach erfolgreicher Anmeldung besteht aus einem Profile-Objekt, das aber nicht gespeichert werden muss. Die folgenden Cmdlets werden automatisch im Kontext dieser Anmeldung ausgeführt, so dass kein Verweis auf das Profile-Objekt verwendet wird.

- **Hinweis** Wird für die Anmeldung ein Azure AD-Benutzerkonto verwendet, kann die Kombination aus Benutzername und Kennwort dem *Credential*-Parameter als *PSCredential*-Objekt übergeben werden.

Beispiel

Der folgende Befehl listet die Namen aller Ressourcengruppen auf, die mit dem bei der Anmeldung verwendeten Konto verknüpft sind:

```
Get-AzureRmResourceGroup | Select-Object ResourceGroupName
```

Für das Lokalisieren von Ressourcen und Ressourcengruppen sind die Cmdlets *Find-AzureRmResource* und *Find-AzureResourceGroup* zuständig. Beide erzeugen keinen Fehler, falls die Ressource nicht existieren sollte.

Beispiel

Der folgende Befehl findet alle Ressourcengruppen, die mit einem bestimmten Schlüsselwertpaar getagt wurden:

```
Find-AzureRmResourceGroup -Tag @{name="Kategorie"; value = "Provisorisch"
}
```

Beispiel

Der folgende Befehl findet alle Ressourcen eines bestimmten Typs, die ein bestimmtes Namensfragment besitzen:

```
Find-AzureRmResource  -ResourceType
Microsoft.ClassicStorage/StorageAccounts -ResourceNameContains Pemo
```

13.4.4 Beispiele aus der Praxis

In diesem Abschnitt werden in loser Reihenfolge ein paar Beispiele für den Umgang mit den ARM-Cmdlets vorgestellt.

Abfragen des VM-Status

Da das *Get-AzureRmVm*-Cmdlet die Eigenschaften einer VM im JSON-Format holt und der aktuelle Status der VM Teil der Eigenschaft *Statuses* ist, muss dieser aus der Eigenschaft extrahiert werden.

Beispiel

Die folgende Befehlsfolge gibt zu allen VMs den Statuswert aus:

```
Get-AzureRmVM  | ForEach-Object {
    # Jetzt mit Ressourcengruppe und VMName Status holen
    Get-AzureRmVm -ResourceGroupName $_.ResourceGroupName -Name $_.Name -
Status -PipelineVariable Vm |
            Select  -ExpandProperty Statuses |
                Where Code  -like "PowerState/*" | Select
@{n="VM";e={$Vm.Name}}, DisplayStatus
}
```

Hochladen einer Datei in ein Speicherkonto

Das Hochladen von Dateien in einen Storage-Account ist eine Formsache, da dazu lediglich das Cmdlet *Set-AzureStorageBlobContent* ausgeführt werden muss.

Beispiel

Das folgende Beispiel lädt eine Musterdatei in einen Blob, der zuvor in einem Speicherkonto angelegt wurde. Für den Namen des Speicherkontos muss ein entsprechender Wert eingesetzt werden.

```
# Interaktive Anmeldung am Azure-Portal
Login-AzureRmAccount

$AzureSub = Get-AzureRmSubscription
if (Test-AzureName -Storage -Name $StorageAccountName)
{
    Write-Verbose "$StorageAccountName existiert bereits."
}
else
{
  $Result = New-AzureStorageAccount -StorageAccountName
$StorageAccountName -Location $Location -Type $StorageType
  if ($Result.OperationStatus -eq "Succeeded") {
    Set-AzureSubscription -SubscriptionName $AzureSub.SubscriptionName[0]
-CurrentStorageAccount $StorageAccountName
    New-AzureStorageContainer -Name $Container -Permission Off
  }
  else
  {
      Write-Warning "Fehler beim Anlegen von $StorageAccountName - Skript
wird beendet."
      exit -1
    }
}

$UploadPath = "C:\Windows\Win.ini"
$Blobname = "IniBlob"

Set-AzureStorageBlobContent -Container $Container -File $UploadPath -Blob
$Blobname

Get-AzureStorageBlob -Blob $BlobName -Container $Container | Select *
```

Bereitstellung einer VM

Die Bereitstellung einer VM ist auch im Zusammenspiel mit dem ARM nicht komplizierter als in der Vergangenheit. Umfangreich und damit etwas komplizierter wird es immer dann, wenn alle benötigten Ressourcen neu angelegt werden. Es sei vorangestellt, dass es fast keinen Sinn mehr ergibt, diese Aufgabe per PowerShell-Cmdlets „zu Fuß“ zu erledigen, da ein Template die deutlich flexiblere Variante darstellt.

Beispiel

Das folgende Beispiel ist etwas umfangreicher, da es insgesamt aus 16 Schritten besteht. Sie finden das komplette Skript als Beispiel exemplarisch wie auch die anderen Beispiele dieses Buches gestaltet sind. Einige Beispiele in diesem Buch sind aus Platzgründen nicht vollständig abgebildet. Sie erhalten aber alle Beispiele als Download (siehe Vorwort).

Das Beispiel zeigt, wie sich eine VM mit allen benötigten Ressourcen wie einem virtuellen Netzwerk, einer öffentlichen IP-Adresse, einem Speicherkonto, einem Laufwerk und einem Betriebssystem neu anlegen lässt. Die Ausführung kann mehrere Minuten in Anspruch nehmen, daher haben Sie etwas Geduld, wenn scheinbar nichts mehr passiert. Am Ende wird die URL der neuen VM ausgegeben, die für einen RDP-Zugriff verwendet werden kann.

Anmeldung

Die Anmeldung erfolgt über das *Login-AzureRmAccount*-Cmdlet:

```
$AzureProfile = Login-AzureRmAccount
```

Schritt 1: Auswahl des Abonnements

Dieser Schritt ist optional. Die Azure-Subscription wird über das *Select-AzureRmSubscription*-Cmdlet ausgewählt:

```
Select-AzureRmSubscription -SubscriptionId
$AzureProfile.Context.Subscription.SubscriptionId -TenantId
$AzureProfile.Context.Tenant.TenantId
```

Schritt 2: Festlegen der Eckdaten der VM

Im nächsten Schritt werden die Eckdaten der VM festgelegt:

```
$RessourceGroupName ="Pemo2017Ressourcen"
$Location = "West Europe"
$VnetName = "StandardVnet"
$IpAddress = "10.0.0.12"
$IpRange = "10.0.0.0/16"
$IpSubNetRange = "10.0.0.0/24"
$Subnetname = "Subnet-1"
$VmSize = "Basic_A0"
$OSSKU = "2012-R2-Datacenter"
$VmName = "PemoVM"
$StorageAccountName = "pemovmstore"
$DomNameLabel  = "pemovm"

# Benutzername und Kennwort
$Username = "PemoAdmin"
$Password = "demo+12345678"
$PasswordSec = $Password | ConvertTo-SecureString -AsPlaintext -Force
$Cred = New-Object -Typename PSCredential -ArgumentList $Username,
$PasswordSec
```

Schritt 3: Anlegen einer Ressourcengruppe
Im nächsten Schritt wird eine Ressourcengruppe („Resource Group“) angelegt, sofern sie noch nicht existiert:

```
try
{
    Get-AzureRmResourceGroup -Name $RessourceGroupName -Location
$Location -ErrorAction Stop | Out-Null
    Write-Verbose "Die Ressourcengruppe $RessourceGroupName gibt es
bereits." -Verbose
}
catch
{
    New-AzureRmResourceGroup -Name $RessourceGroupName -Location
$Location -Verbose
}
```

Schritt 4: Anlegen eines Speicherkontos
Im nächsten Schritt wird ein neues Speicherkonto („Storage Account“) angelegt. Für den Namen gelten speziellen Regeln, unter anderem darf der Name nur aus Kleinbuchstaben bestehen. Damit es nicht zu einem Fehler kommt, wenn der Name bereits existieren sollte, wird über eine kleine Schleife ein Name gesucht, der noch nicht existiert:

```
$StorageNameIndex = 0
do
{
    $StorageNameIndex++
    $StorageAccountName = "vmstore{0:00}" -f $StorageNameIndex
} until ((Get-AzureRmStorageAccountNameAvailability -Name
$StorageAccountName).NameAvailable)

New-AzureRmStorageAccount -Name $StorageAccountName -ResourceGroupName
$RessourceGroupName  -Location $Location -SkuName Standard_LRS
$StorageAccount = Get-AzureRmStorageAccount -ResourceGroupName
$RessourceGroupName -StorageAccountName $StorageAccountName
```

Schritt 5: Uri für Vhd im Storage holen
Im nächsten Schritt wird die Uri für die zu speichernde Vhd-Datei geholt:

```
$DiskNameOS = $VmName + "_DiskOS"
$VhdUri = $StorageAccount.PrimaryEndpoints.Blob + "vhds/$DiskNameOS.vhd"
```

Schritt 6: Image-Dateien holen
Im nächsten Schritt wird die bereits vorhandene Image-Datei für Windows Server 2012 R2 im Azure-Portal ausgewählt:

```
$VmImages = Get-AzureRmVMImage -Location $Location -PublisherName
"MicrosoftWindowsServer" -Offer "WindowsServer" -Skus $OsSKU | Sort-
Object -Descending -Property PublishedDate
```

Schritt 7: Public IP anlegen
Im nächsten Schritt wird eine öffentliche IP-Adresse angelegt:

```
$PublicIP = New-AzureRmPublicIpAddress -Name "$VmName`_Nic1" -
ResourceGroupName $RessourceGroupName -DomainNameLabel $DomNameLabel -
Location $Location -AllocationMethod Dynamic –Force
```

Schritt 8: Subnet erstellen
Im nächsten Schritt wird ein Subnet angelegt:

```
$Subnet1 = New-AzureRmVirtualNetworkSubnetConfig -Name $SubnetName -
AddressPrefix $IPSubNetRange
```

Schritt 9: Virtual Network anlegen
Im nächsten Schritt wird ein virtuelles Netzwerk angelegt:

```
try
{
  $Vnet = Get-AzureRmVirtualNetwork -Name $VnetName -ResourceGroupName
$RessourceGroupName -ErrorAction Stop
  Write-Verbose "$VNET gibt es bereits..." –Verbose
}
catch
{
  $Vnet = New-AzureRmVirtualNetwork -Name $VnetName -ResourceGroupName
$RessourceGroupName -Location $Location -AddressPrefix $IPRange -Subnet
$Subnet1 –Force
}
$Subnet = Get-AzureRmVirtualNetworkSubnetConfig -VirtualNetwork $Vnet
```

Schritt 10: Netzwerkadapter anlegen
Im nächsten Schritt wird der Netzwerkadapter angelegt:

```
$NIC = New-AzureRmNetworkInterface -Name "$VmName`_Nic1" -Location
$Location -ResourceGroupName $RessourceGroupName -SubnetId
$Vnet.Subnets[0].Id -PublicIpAddressId $PublicIP.Id –Force
```

Schritt 11: VM-Konfiguration anlegen
Damit ist die VM fast fertig. Zum Schluss werden alle Einstellungen in einer VM-Konfiguration zusammengefasst:

```
$NewVM = New-AzureRmVMConfig -Name $VmName -VMSize $VmSize
$NewVM = Add-AzureRmVMNetworkInterface -VM $NewVM -Id $NIC.Id
```

Schritt 12: OS der VM zuordnen
Im nächsten Schritt wird der VM-Konfiguration ein Betriebssystem zugeordnet:

```
Set-AzureRmVMOperatingSystem -Windows -VM $NewVM -ProvisionVMAgent -
EnableAutoUpdate:$false -ComputerName $VmName -Credential $Cred
```

Schritt 13: Image der VM zuordnen
Im nächsten Schritt wird der VM-Konfiguration das bereits ausgewählte Image zugeordnet:

```
Set-AzureRmVMSourceImage -VM $NewVM -PublisherName
$VmImages[0].PublisherName -Offer $VmImages[0].Offer -Skus
$VmImages[0].Skus -Version $VmImages[0].Version
```

Schritt 14: Laufwerkseigenschaften setzen
Im nächsten Schritt wird per *Set-AzureRmVMOSDisk*-Cmdlet das Laufwerk konfiguriert. Per *CreateOption*-Parameter wird festgelegt, dass das Laufwerk aus einem Image angelegt werden soll.

```
Set-AzureRmVMOSDisk -VM $NewVM -Name $DisknameOS -VhdUri $VhdUri -Caching
ReadWrite -CreateOption FromImage
```

Schritt 15: VM anlegen
Im letzten Schritt kann die neue VM per *New-AzureRmVM*-Cmdlet endlich angelegt werden. Dieser Vorgang kann einige Minuten dauern (Abb. 13.11).

```
New-AzureRmVM -ResourceGroupName $RessourceGroupName -Location $Location
-VM $NewVM -Verbose
```

Wurde die VM angelegt, wird sie per *Start-AzureVmRm*-Cmdlet gestartet:

```
Start-AzureRmVM -Name $VmName -ResourceGroupName $RessourceGroupName  -
Verbose
```

Die für den RDP-Zugang erforderliche URL kann aus den Eckdaten der VM abgeleitet werden:

```
$AzureRmVM = Get-AzureRmVm -ResourceGroupName $RessourceGroupName -Name
$VmName
$RdpURL = "$($PublicIP.DnsSettings.Fqdn):3389"
```

13.4.5 Eine virtuelle Maschine über ein Template anlegen

Deutlich einfacher geschieht das Anlegen von virtuellen Computern über ein Template, vor allem dann, wenn das Template bereits vorhanden ist, wie es bei den QuickStart-Templates der Fall ist, die zum Beispiel auf GitHub angeboten werden. In diesem Fall

Abb. 13.11 Die per PowerShell-Skript angelegte VM wird im Azure-Portal angezeigt

beschränkt sich das Anlegen der VM auf das Festlegen der Parameter und dem Aufruf von *New-AzureRmResourceGroupDeployment*.

Beispiel

Das folgende Beispiel legt eine Linux-VM in der angegebenen Ressourcengruppe an. Damit es funktioniert, müssen eine Reihe von Angaben angepasst werden, wie zum Beispiel der Name der Ressourcengruppe:

```
$ResourceGroupName = "Standardgruppe"
$TemplateURL = https://raw.githubusercontent.com/Azure/azure-quickstart-
templates/master/101-vm-simple-linux/azuredeploy.json

$DeploymentSettings = @{
  ResourceGroupName = $ResourceGroupName
  TemplateUri = $TemplateURL
  TemplateParameterObject = @{
    adminUserName = 'PemoAdmin'
    adminPassword = 'demo+123'
    dnsLabelPrefix = 'pm-arm'
  }
  Mode = 'Complete'
  Force = $true
  Name = 'TestDeployment'
}

New-AzureRmResourceGroupDeployment @DeploymentSettings -Verbose
```

Die Parameter werden über ein Splatting übergeben.

Für selber erstellte Templates empfiehlt sich die Verwendung des *Test-AzureRm-ResourceGroupDeployment*-Cmdlets. Da dieses aber die Parameter Name und Force nicht unterstützt, müssen beide in der Parameter-Hashtable auskommentiert werden.

13.4.6 Azure und DSC

Die *Desired State Configuration* (DSC) spielt auch im Azure-Portal eine Rolle, zum Beispiel für die Konfiguration von VMs. Die Grundlage ist ein Azure-Automation-Konto, das zuvor angelegt werden muss. Die Vorgehensweise ist grundsätzlich einfach. Die wichtigste Kleinigkeit besteht darin, dass sobald Konfigurationsdaten im Spiel sind, die Konfiguration nicht im Portal, sondern per PowerShell-Cmdlet Start-AzureRmAutomati onDscCompilationJob kompiliert werden muss, so dass sie anschließend einer VM zugeordnet werden kann.

Für das folgende kleine Beispiel müssen drei Voraussetzungen erfüllt sein:

1. Ein Azure-Konto.
2. Ein Azure-Automation-Konto, das innerhalb des Azure-Kontos angelegt wurde.
3. Eine bereits vorhandene Windows-VM (theoretisch würde es auch mit einer Linux-VM funktionieren).

Der erste Schritt besteht im Anlegen einer DSC-Konfiguration, die absichtlich sehr einfach gehalten ist. Ihre Aufgabe besteht darin, ein Verzeichnis mit einer Datei anzulegen.

Der Name des Verzeichnisses und der Inhalt der Datei werden über Konfigurationsdaten festgelegt.

Die Konfiguration heißt *CreateDirectoryWithFile* und liegt als Ps1-Datei vor. Sie ist wie folgt aufgebaut:

```
<#
 .Synopsis
 Eine kleine Test-Konfiguration fuer Azure DSC
#>
configuration CreateDirectoryWithFile
{
  Import-DSCResource -ModuleName PSDesiredStateConfiguration

  node $AllNodes.NodeName
  {
    file Dir1
    {
       Ensure = "Present"
        DestinationPath = "C:\$($Node.DirName)"
        Type = "Directory"
    }

    file file1
    {
        Ensure = "Present"
        DestinationPath = "C:\$($Node.DirName)\DSC.txt"
        Contents = $Node.FileContent
        Type = "File"
    }
  }
}
```

Die Konfigurationsdaten werden entweder in einer lokalen Psd1-Datei abgelegt oder sind Teil des Bereitstellungsskripts. Sie kommen in Kürze ins Spiel.

Im nächsten Schritt wird die Ps1-Datei mit der Konfiguration in das Azure-Portal geladen. Wählen Sie dazu das bereits angelegte Automations-Konto, selektieren Sie dort **DSC-Konfigurationen** und danach **+ Konfiguration hinzufügen** (Abb. 13.12).

Wählen Sie im nächsten Schritt die Ps1-Datei aus, die die Konfiguration *CreateDirectoryWithFile* enthält (Abb. 13.13).

An dieser Stelle kann noch nicht viel schiefgehen, so dass die Konfiguration anschließend in der Liste der verfügbaren Konfigurationen erscheint (Abb. 13.14).

Jetzt kommt der entscheidende Schritt: Die Konfiguration muss in eine Mof-Datei kompiliert werden. Solange die Konfiguration keine Konfigurationsdaten verwendet, wählen Sie den Button **Kompilieren** in den Eigenschaften der Konfiguration, die nach ihrer Auswahl eingeblendet werden. Würden Sie den Button klicken, würde die Umsetzung

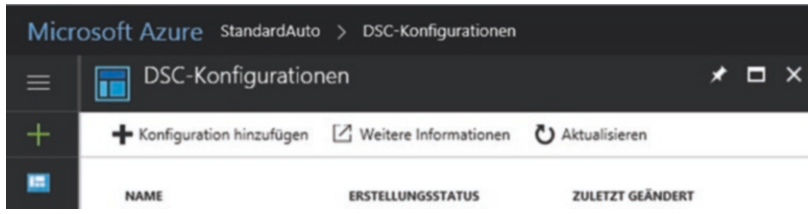

Abb. 13.12 Eine neue DSC-Konfiguration wird angelegt

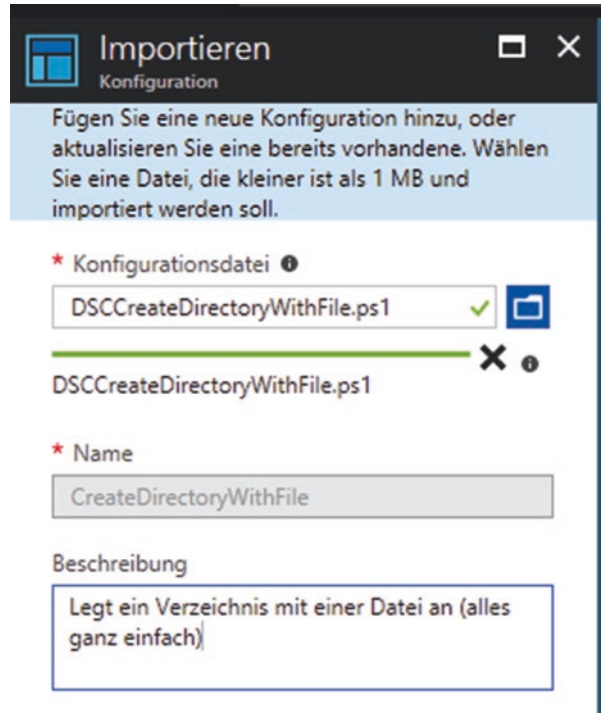

Abb. 13.13 Die DSC-Konfiguration wird als Ps1-Datei in das Azure-Portal geladen

Abb. 13.14 Die Konfiguration steht im Rahmen des Automations-Konto zur Verfügung

kurz danach mit einer relativ nichtssagenden Meldung anhalten. Die Kompilierung muss per PowerShell und dem Cmdlet *Start-AzureRmAutomationDscCompilationJob* aus dem *AzureRM.Automation*-Modul lokal durchgeführt werden. Bei dieser Gelegenheit kommen auch die Konfigurationsdaten im Spiel.

Beispiel

Die folgende Befehlsfolge kompiliert eine DSC-Konfiguration, die bereits hochgeladen wurde. Eine erfolgte Anmeldung an das Azure-Portal wird vorausgesetzt.

```
<#
 .Synopsis
 Azure DSC per Cmdlets
#>

$ConfigData = @{
  GeneralData = @{
        Message = "Alles klar mit Azure DSC!"
  }

    AllNodes = @(
        @{
          NodeName = "TestVM2"
          DirName = "TestDir5678"
          FileContent = "Pemo-DSC was here again!"
          }
  )
}

$CompilationJob = Start-AzureRmAutomationDscCompilationJob -
ResourceGroupName Standardgruppe `
 -AutomationAccountName Standardauto -ConfigurationName
CreateDirectoryWithFile -ConfigurationData $ConfigData

while($CompilationJob.EndTime -eq $null -and $CompilationJob.Exception -
eq $null) {
    $CompilationJob = $CompilationJob | Get-
AzureRmAutomationDscCompilationJob
    Start-Sleep -Seconds 3
}
$CompilationJob | Get-AzureRmAutomationDscCompilationJobOutput -Stream
Any
```

Ging alles gut, wird der Kompilierungsauftrag als abgeschlossen angezeigt und die Knotenkonfiguration steht auf einem Pull-Server zur Verfügung (Abb. 13.15).

Zum Schluss muss der Pull-Server mit der VM verbunden werden. Das geschieht nicht automatisch. Wechseln Sie im Azure-Portal zurück zum Automations-Konto und wählen Sie **DSC-Knoten** und dort **+ Azure-VM hinzufügen**. Im nächsten Schritt wählen Sie die VM in zwei Teilschritten aus. Selektieren Sie zuerst den oberen Kasten **Virtuelle Computer** und wählen Sie dort die VM aus. Nach der Auswahl der VM und dem Bestätigen mit **OK** wird der Kasten wieder zugeklappt. Wählen Sie im zweiten Schritt **Registrierung**. Wählen Sie den Namen der Konfiguration aus, die über ihren Namen bereits der zugeordnet ist, übernehmen Sie die Voreinstellungen und bestätigen Sie auch diese Auswahl mit **OK**. Auch dieser Kasten wird danach wieder zugeklappt. Klicken Sie jetzt auf **Erstellen**. Dadurch wird die DSC-Konfiguration bei der ausgewählten VM registriert (Abb. 13.16).

Anschließend müssen Sie 15 Minuten warten, bis die Konfiguration auf die VM angewendet wird. Ging alles gut, sollte ein Verzeichnis mit einer Datei und dem über die Konfigurationsdaten festgelegten Inhalt angelegt worden sein (Abb. 13.17). Azure DSC ist eine flexible Angelegenheit, die zudem im strategischen Fokus von Microsoft liegt. Wenn Sie diese Zeilen lesen, dürften weitere wichtige Neuerungen, die die zugrunde liegende Architektur von DSC betreffen, verfügbar sein. Vor dem Einsatz muss allerdings die Kostenfrage geklärt werden, denn die Dienstleistung ist nicht kostenlos.

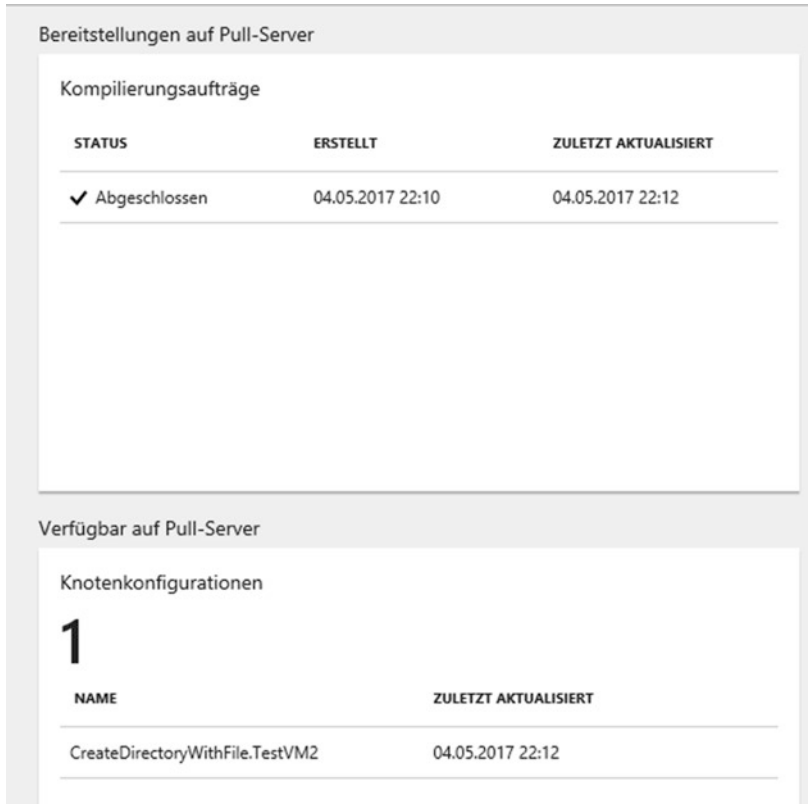

Abb. 13.15 Die Knotenkonfiguration steht auf einem Pull-Server bereit

13.5 Custom Script Extension als Alternative zu DSC

Jeder VM kann genau eine *Custom Script Extension* in Gestalt eines Agenten zugeordnet werden, der unabhängig von der VM ausführt. Darüber lässt sich zum Beispiel ein PowerShell-Skript angeben, dass mit dem nächsten Booten der VM automatisch ausgeführt wird. Das Skript kann auf unterschiedliche Weise zur Verfügung gestellt werden. Im einfachsten Fall wird es im Azure-Portal beim Hinzufügen oder Konfigurieren der Erweiterungen hochgeladen. Es kann über ein Template bereitgestellt werden. In diesem Fall kann sich das Skript zum Beispiel in einem Storage Account befinden. Es kann natürlich auch per PowerShell-Cmdlet festgelegt werden. Zuständig ist das *Set-AzureRmVMCustomScriptExtension*-Cmdlet aus dem *AzureRm.Compute*-Modul.

Auch wenn einer VM jeweils nur eine Erweiterung zugeordnet werden kann, können über die Parameter *FileName* bzw. *FileUri* auch *Set-AzureRmVMCustomScriptExtension*-Cmdlets auch mehrere Skripts festgelegt werden, die durch die Erweiterung in den lokalen Downloads-Ordner der VM geladen werden (*C:\Packages\Plugins\Microsoft.Compute.CustomScriptExtension\1.8\Downloads*). In diesem Fall wird das auszuführende Skript

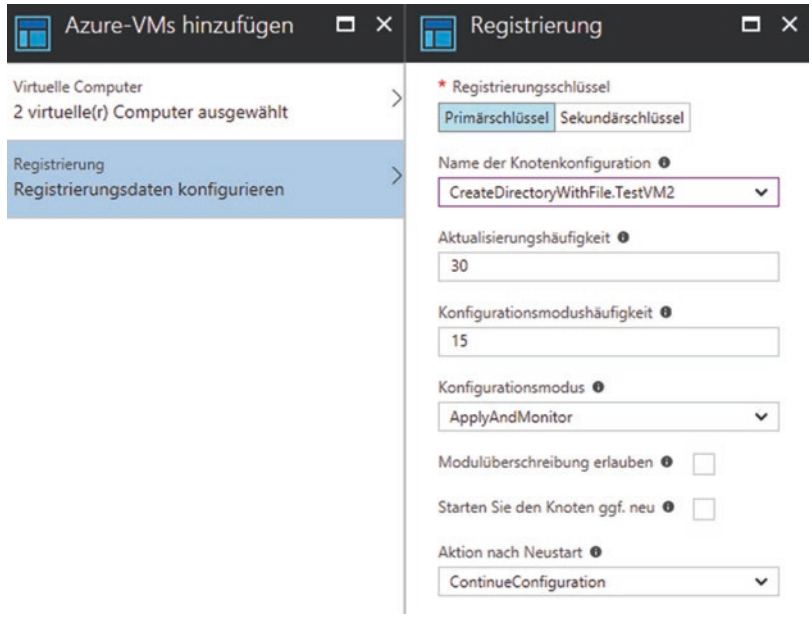

Abb. 13.16 Die DSC-Konfiguration wird einer VM zugeordnet

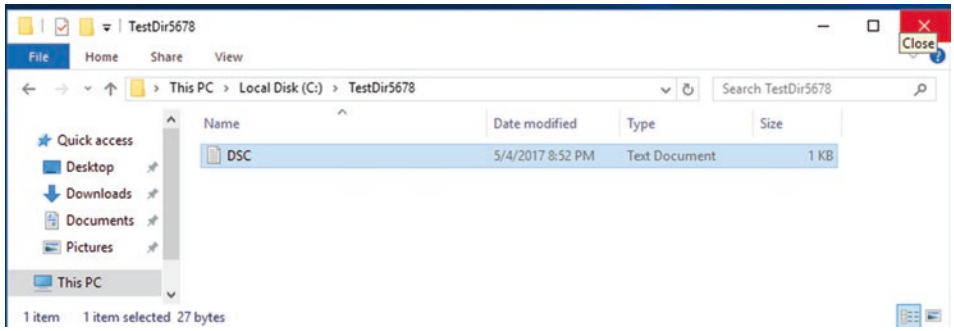

Abb. 13.17 Die DSC-Konfiguration wurde in der VM umgesetzt

über den *Run*-Parameter festgelegt. Über den *FileName*-Parameter muss nicht unbedingt ein Ps1-Skript angegeben werden, es kann zum Beispiel auch eine Exe-Datei sein.

Beispiel

Das folgende Beispiel führt eine Ps1-Datei in einer VM über eine Custom Script Extension aus. Die Ps1-Datei lädt eine Exe-Datei aus einem Azure-Speicherkonto (es könnte natürlich auch ein AWS-Speicherkonto oder eine Ftp-Adresse sein) und führt diese aus. Das kleine Beispiel demonstriert damit, wie sich eine Anwendung in einer VM installieren lässt. Am Ende wird die Exe-Datei über *Invoke-Expression* ausgeführt, da der etwas naheliegendere Aufruf per *Start-Process* zu einem „Access Denied“ führt. Das Skript ist wie folgt aufgebaut:

```
<#
 .Synopsis
 Installiert eine Anwendung über den Download einer Exe-Datei
#>

Set-StrictMode -Version Latest

$PackageUri =
"https://standardspeicher.blob.core.windows.net/packages/SumatraPDF-
3.1.2-64-install.exe"
$PackageArgs = "/S"

$TempPath = $env:TEMP
$LogPath = Join-Path -Path $TempPath -ChildPath "AzPackageInstall.log"
$PackagePath = Join-Path -Path $TempPath -ChildPath (Split-Path -Path
$PackageUri -Leaf)

Add-Content -Path $LogPath -Value ("$PackagePath wurde um {0:HH:mm} in
$TempPath heruntergeladen." -f (Get-Date))

# Herunterladen der Exe-Datei
Invoke-WebRequest -Uri $PackageUri -OutFile $PackagePath

Invoke-Expression "$PackagePath /S"

Add-Content -Path $LogPath -Value ("$PackagePath wurde um {0:HH:mm}
ausgeführt" -f (Get-Date))
```

Das Skript, das die Custom Script Extension hinzufügt, ist wie folgt aufgebaut:

```
<#
 .Synopsis
 Umgang mit Custom Script Extensions
#>

$Username = "<username>
$PwSec = "<kennwort>" | ConvertTo-SecureString -AsPlainText -Force
$AzureCred = [PSCredential]::new($Username, $PwSec)
Login-AzureRmAccount -Credential $AzureCred | Out-Null

# Abfragen der vorhandenen Erweiterung
$VMName = "TestVM"
$ResourceGroupname = "Standardgruppe"
$ExtName = "InstallAppForVm"
$Ps1FileName = "Azure_InstallAppForVM.ps1"

# Anlegen einer neuen CustomScriptExtension
$FileUri =
"https://standardspeicher.blob.core.windows.net/ps1blob/Azure_InstallAppF
orVM.ps1"
$Location = "westeurope"
$StorageAccountName = "standardspeicher"

# Anlegen der Custom Script Extension
Set-AzureRmVMCustomScriptExtension `
    -ResourceGroupName $ResourceGroupname `
    -VMName $VMName `
    -Location $Location `
    -FileUri $FileUri `
    -Run $Ps1FileName `
    -Name $ExtName

Get-AzureRmVMCustomScriptExtension -VMName $VMName -ResourceGroupName
$ResourceGroupname -Name InstallAppForVm
```

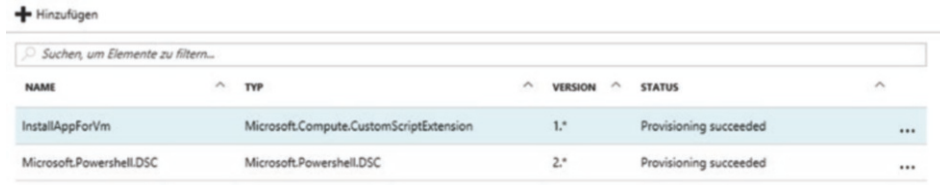

Abb. 13.18 Die Custom Script Extension wurde in der Azure VM bereitgestellt

Eine ausführliche Beschreibung der Custom Script Extensions (Abb. 13.18) finden Sie auf *docs.microsoft.com* unter der folgenden Adresse: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/extensions-customscript.

```
Set-AzureRmVMCustomScriptExtension `
    -ResourceGroupName $ResourceGroupname `
    -VMName $VMName `
    -Location $Location `
    -FileUri $FileUri `
    -Run "Azure_InstallAppForVM.ps1" `
    -Name "InstallAppForVm"
```

13.6 Zusammenfassung

Um die stetig steigende Zahl an Diensten im Rahmen der Azure-Plattform besser auf eine durchdachte Art und Weise anbieten zu können, hat Microsoft in den vergangenen Jahren nicht nur die Benutzeroberfläche im Azure-Portal, sondern auch die Art und Weise der Bereitstellung von Diensten auf den *Azure Ressource Manager* (ARM) umgestellt. Mit einher geht eine neue Azure PowerShell, die einen neuen Satz an Modulen mit Cmdlets enthält, die den ARM ansprechen. Mit dem Umstand, dass Ressourcen über Vorlagen definiert und diese parametrisiert werden können, geht eine deutliche Steigerung der Flexibilität einher. Benötigt eine Schulungsabteilung einen bestimmten Satz an VMs, die mit einer bestimmten Software bestückt werden, wird lediglich das passende Template benötigt, das in der Regel über das GitHub-Portal zur Verfügung stehen dürfte. Das erspart nicht nur ein stundenlanges Hindurchklicken durch die Benutzeroberfläche des Azure-Portals oder umfangreiche PowerShell-Skripte, sondern stellt auch sicher, dass die gewünschte Konfiguration entsteht. Mit dem ARM zieht Microsoft in Punkto Konfiguration mit Amazon Webservices gleich.

Debugging für etwas Fortgeschrittene

14

Zusammenfassung

Das Debuggen eines PowerShell-Skripts ist grundsätzlich keine komplizierte Angelegenheit, insbesondere in der PowerShell ISE und bei Visual Studio Code mit installierter PowerShell-Erweiterung. Einfach einen Haltepunkt setzen und das Skript starten. Erreicht die Ausführung die Zeile mit dem Haltepunkt, kann das Skript ab diesem Zeitpunkt Befehl für Befehl ausgeführt werden, entweder über die Funktionstasten bei ISE und Code oder über die einfachen Debugger-Kommandos, die im Debug-Modus über die Tastatur eingegeben werden. In diesem Kapitel geht es um die etwas fortgeschrittenen Themen wie Haltepunkte, die von einer Bedingung abhängen, das Debuggen von Remote-Skripten, Workflows und Runspaces, die theoretisch Teil einer beliebigen Anwendung sein können.

Die PowerShell-Hilfe gibt unter dem Topic „about_debuggers“ eine ausführliche Übersicht zum Thema Debugger. Dort wird zum Beispiel auf jene Variablen eingegangen, deren Wert beim Debuggen nicht direkt abgefragt werden können. Was im Grundlagenlehrgang noch als recht speziell erschienen wäre, erscheint in diesem Lehrgang als ein sinnvoller Überblick über ein Kernthema der PowerShell. Weitere Beispiele gibt es in der Hilfe zu den einzelnen Debugger-Cmdlets wie *Set-PSBreakPoint*.

14.1 Unsichtbare Variablen beim Debuggen

Nicht jede der automatischen Variablen kann während des Debuggens direkt abgefragt werden. Die Ausnahmen sind *Args*, *Input*, *MyInvovation* und *PSBoundParameters*. Insbesondere bei letzterer Variable wird wohl jeder bereits vor der Situation gestanden haben, dass ihr Wert beim Debuggen einfach leer blieb. Die Lösung besteht darin, den Wert dieser Variablen einer anderen Variablen zuzuweisen, über die ein Wert abgefragt werden kann.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_14

Beispiel

Das folgende Beispiel verwendet Hilfsvariablen, um die Werte einiger automatischer Variablen während des Debuggens abfragen zu können.

```
param([Int]$Anzahl = 10)
$Cmd = $MyInvocation
$Paras = $PSBoundParameters
1..$Anzahl | ForEach {
  $PipeVar = $_
  "Durchlauf Nr. $_"
}
```

14.2 Der Debug-Modus im Überblick

Der Debug-Modus ist ein spezieller Modus der PowerShell. Er wird durch den Zusatz „[DBG]“ beziehungsweise „[WFDBG]“ für Workflows im Prompt angezeigt. Die Variable *PSPromptLevel* wird um eins erhöht. Im Debug-Modus enthält die Variable *PSDebugContext* alle Informationen über das Skript, das aktuell debuggt wird. Durch Abfragen der Variablen lässt sich feststellen, ob ein Skript im Debug-Modus ausgeführt wird.

14.3 Über das Wesen eines Haltepunktes

Auch ein Haltepunkt ist bei der PowerShell natürlich ein Objekt. Das *Get-PSBreakPoint*-Cmdlet liefert alle aktuell gesetzten Haltepunkte in Gestalt von *LineBreakPoint*-Objekten. Sie verfügen über die Eigenschaften *Action*, *Column*, *Enabled*, *HitCount*, *Id*, *Line* und *Script*. Ein Haltepunkt bezieht sich auf eine bestimmte Zeile, eventuell ergänzt durch eine Spaltennummer, einen bestimmten Befehl, eine bestimmte Variable oder allgemein auf einen Ausdruck.

- **Hinweis** Haltepunkte existieren nur für die Dauer einer PowerShell-Sitzung. Wie jedes Objekt kann auch ein Breakpoint-Objekt per *Export-CliXml*-Cmdlet in eine XML-Datei gespeichert werden. Das erneute Importieren der Haltepunkte ist aber nur über einen kleinen Umweg möglich, indem die per *Import-CliXml* eingelesenen Haltepunkte im Rahmen einer Wiederholung per *ForEach-Object* per *Set-BreakPoint*-Cmdlet neu angelegt werden.

14.3.1 Allgemeine Haltepunkte setzen

Ein allgemeiner Haltepunkt wird aktiv, wenn die Ausführung eine bestimmte Zeile und gegebenenfalls auch eine bestimmte Spalte erreicht hat.

Beispiel

Der folgende Befehl setzt in dem Skript „Test.ps1“ in Zeile 3 einen Haltepunkt.

```
Set-PSBreakPoint -Script .\Test.ps1 -Line 3
```

14.3.2 Haltepunkt entfernen und deaktivieren

Damit ein Skript mit einem gesetzten Haltepunkt wieder regulär ausgeführt werden kann, müssen der oder die Haltepunkte über die Cmdlets *Remove-PSBreakPoint* und *Disable-PSBreakPoint* entweder entfernt oder deaktiviert werden.

14.3.3 Haltepunkte für einen Befehl setzen

Soll ein Haltepunkt nur dann aktiv werden, wenn ein bestimmtes Command ausgeführt wird (ein Cmdlet, eine Function oder auch ein Skript), muss der Name des Befehls auf den Command-Parameter folgen. Wann immer das Command ausgeführt wird, geht die Ausführung in den Debug-Modus über.

Beispiel

Der folgende Befehl setzt den Haltepunkt auf das *Start-Sleep*-Cmdlet.

```
Set-PSBreakPoint -Script .\Test.ps1 -Command "Start-Sleep"
```

14.3.4 Haltepunkte für eine Variable setzen

Ein Haltepunkt kann auch von einer Variablen abhängig gemacht werden. Dazu wird bei *Set-PSBreakPoint* der Name der Variablen mit dem Variable-Parameter angegeben. Wird die Variable lesend oder schreibend angesprochen, wird der Haltepunkt aktiv.

Beispiel

Der folgende Befehl setzt einen Haltepunkt auf die Variable *Anzahl*.

```
Set-PSBreakPoint -Script .\Test.ps1 -Variable Anzahl
```

Soll sich der Haltepunkt gezielt auf einen Lese- oder Schreibzugriff einer Variable beziehen, kommt beim *Set-PSBreakPoint*-Cmdlet der *Mode*-Parameter ins Spiel. Über ihn wird festgelegt, ob der Haltepunkt nur bei einem Lese- oder Schreibzugriff oder in beiden Fällen (dies ist der Default) aktiv werden soll.

14.3.5 Haltepunkte mit einer Bedingung verknüpfen

Soll ein Haltepunkt nicht immer, sondern nur dann, wenn eine bestimmte Bedingung erfüllt ist, aktiv werden, kommt der *Action*-Parameter des *Set-PSBreakPoint*-Cmdlet ins Spiel. Über diesen Parameter wird allgemein eine Aktion als Scriptblock angegeben, die mit jedem Erreichen des Haltepunktes ausgeführt wird. Soll der Haltepunkt daher nur dann aktiv werden, wenn eine Bedingung erfüllt ist, muss die Bedingung innerhalb des Scriptblocks geprüft werden und für den Fall, dass sie erfüllt ist, muss der Haltepunkt über den *break*-Befehl aktiviert werden.

Beispiel

Der folgende Befehl setzt einen Haltepunkt, der nur dann aktiv wird, wenn die Variable i größer als 5 ist.

```
Set-PSBreakPoint –File .\Test.ps1 –Line 5 –Action { if ($i –gt 5) { break
} }
```

14.3.6 Ein Skript ohne Haltepunkte debuggen

Ein Skript kann seit der Version 5.0 debuggt werden, ohne einen Haltepunkt setzen zu müssen. Dazu wird es während der Ausführung über die Tastenkombination [Strg]+[Break] unterbrochen. In der ISE übernimmt diese Funktion der Menübefehl **Debuggen|Alle trennen** oder die Tastenkombination [Strg]+[B]. Die Tastenkombination [Strg]+[Break] war bei der ISE bereits belegt, denn sie bewirkt den Abbruch der Skriptausführung. Das Skript wird in den Debug-Modus geschaltet und hält bei dem Skriptbefehl an, der als nächstes an der Reihe wäre. Das Unterbrechen funktioniert auch für Skripte, die im Rahmen von PowerShell-Remoting auf einem anderen Computer ausgeführt werden.

▶ **Hinweis** Die Unterbrechung eines Skripts mit dem Umschalten in den Debug-Modus funktioniert nur für reguläre Skriptbefehle. Wartet das Skript auf den Rückgabewert eines Methoden-Aufrufs einer .NET-Assembly oder auf die Beendigung eines externen Programms, hat eine Unterbrechungsanforderung keine Wirkung.

14.4 Remote-Debugging von Skripten

Remote-Debugging bedeutet in diesem Zusammenhang, dass ein Skript, das auf einem Remote-Computer ausgeführt werden soll, auf dem lokalen Computer im Rahmen einer Remote-Session debuggt wird. Der Komfortgewinn besteht darin, dass das Skript nicht mehr auf den lokalen Computer kopiert werden muss, um es debuggen zu können. Voraussetzung ist, dass zuvor mit dem Remote-Computer eine Remote-Session angelegt wurde, und dass sich das Skript auf dem Remote-Computer befindet. Das Remote-Debugging, das mit der Version 4.0 eingeführt wurde, unterscheidet sich grundsätzlich nicht vom Debuggen eines lokalen Skripts.

Im ersten Schritt wird per *Enter-PSSession* eine Verbindung mit dem Remote-Computer hergestellt. Im nächsten Schritt wird per *Set-PSBreakPoint*-Cmdlet ein Haltepunkt gesetzt und das Skript gestartet. Wie bei der lokalen Ausführung wird der Haltepunkt aktiv und das Skript wird über die Debugger-Kommandos debuggt.

▶ **Hinweis** Ein kleiner Unterschied zwischen dem lokalen Debuggen und dem Remote-Debugging besteht darin, dass die Ausgabe eines Befehls erst mit dem Ausführen des nächsten Befehls übertragen wird und daher nicht unmittelbar nach der Ausführung des Befehls in der Konsole erscheint.

14.4.1 Haltepunkte über Invoke-Command setzen

Wird ein Skript auf einem Remote-Computer mit einem Haltepunkt über das *Invoke-Command*-Cmdlet ausgeführt, wird die Session automatisch unterbrochen („disconnected"). Mit dem Betreten der Session über das *Enter-PSSession*-Cmdlet kann das Debuggen des Skripts fortgesetzt werden.

Im ersten Schritt wird per *New-PSSession*-Cmdlet eine neue Session zu einem Remote-Computer angelegt. Im zweiten Schritt wird über *Invoke-Command* das *Set-PSBreakPoint*-Cmdlet ausgeführt, um einen Haltepunkt in dem Remote-Skript zu setzen. Im dritten Schritt wird das Skript durch einen erneuten Aufruf von *Invoke-Command* ausgeführt. Mit dem Erreichen des Haltepunktes wird die Verbindung unterbrochen und die Session erhält den Status „Disconnected". Die *Availability*-Eigenschaft (sie muss per *Select-Object* abgefragt werden) erhält den Wert „Remote-Debug". Im vierten Schritt wird die Session per *Enter-PSSession* betreten. Dadurch wird die Debug-Sitzung aktiviert und das Skript kann mit den Debugger-Kommandos debuggt werden.

14.5 Einen Workflow debuggen

Mit der Version 4.0 der PowerShell wurde die Möglichkeit eingeführt, auch Workflows debuggen zu können. Dazu muss per *Set-PSBreakPoint*-Cmdlet ein Haltepunkt in der Workflow-Skriptdatei gesetzt werden. Alles Weitere verläuft genauso wie in einem regulären Skript.

14.6 Runspaces debuggen

Ein Runspace ist der Rahmen, in dem jede PowerShell-Befehlsfolge ausgeführt wird. Mit jedem Start der PowerShell-Konsole, der ISE oder jeder anderen PowerShell-Hostanwendung wird automatisch ein Runspace angelegt. Dieser Runspace, der immer da ist, ist der Default-Runspace. Runspaces gibt es daher bereits seit der Version 1.0, da jeder Befehl nur in einem Runspace ausgeführt werden kann. Bis zur Version 5.0 war es aber in erster Linie ein

Thema für die PowerShell-Insider und für Entwickler. Mit der Version 5.0 wurde mit *Get-Runspace* ein Cmdlet eingeführt, mit dem sich ein Runspace direkt ansprechen lässt. Das zweite Cmdlet, das ebenfalls erst seit Version 5.0 zur Verfügung steht, ist das *Debug-Runspace*-Cmdlet. Mit seiner Hilfe lässt sich ein lokaler Runspace oder ein Remote-Runspace debuggen.

Das Debuggen eines Remote-Runspaces bedeutet ein Skript debuggen zu können, das von einem anderen PowerShell-Skript oder von einer Host-Anwendung in einem eigenen Runspace ausgeführt wird. Für das Debuggen des Standard-Runspaces ist diese Technik nicht gedacht, denn dafür gibt es den regulären Debugger. Im Mittelpunkt steht das *Enter-PSHost*-Cmdlet, mit dem sich eine PowerShell-Host-Anwendung von einer anderen PowerShell-Hostanwendung ähnlich betreten lässt, wie per *Enter-PSSession* eine Session auf einem Remote-Computer betreten wird (technisch unterscheiden sich beide Verfahren grundlegend, das *Enter-PSHost*-Cmdlet basiert auf sogenannten „Named Pipes“, die eine Eigenschaft des Betriebssystems sind). Wurde der Host betreten, listet das *Get-Runspace*-Cmdlet alle zur Auswahl stehenden Runspaces auf. Über das *Debug-Runspace*-Cmdlet wird ein über seine Id ausgewählter Runspace in den Debug-Modus versetzt, so dass das in diesem Runspace ausführende Skript mit den üblichen Kommandos des Debuggers debuggt werden kann. Ein typisches Szenario ist ein PowerShell-Skript, das in der ISE ausgeführt wird. Das Skript legt einen neuen Runspace an, lädt ein Skript in diesem Runspace und führt es aus. Von einem PowerShell-Konsolenhost wird das *Enter-PSHost*-Cmdlet mit der Id des ISE-Prozesses ausgeführt. Ein *Get-Runspace* listet die Ids des durch das ISE-Skript angelegten Runspaces aus. Ein *Debug-Runspace*-Cmdlet setzt diesen Runspace in den Debugmodus, so dass er im Konsolenhost mit den Debugkommandos debuggt werden kann.

Beispiel

Das folgende Beispiel für das Debuggen eines Runspaces ist zwangsläufig etwas umfangreicher und besteht aus mehreren Schritten.

Ausgangspunkt ist ein kleines Skript, das etwas in mehreren Schritten erledigt. Die Beschreibung ist absichtlich so allgemein gehalten, da es keine Rolle spielt, was das Skript im Detail macht.

In der PowerShell ISE wird ein Skript ausgeführt, das das erwähnte Skript in einem neuen Runspace ausführt.

```
# Ein Skript in einem Runspace ausführen

$Ps = [PowerShell]::Create()
$Rs = [RunspaceFactory]::CreateRunspace()
$Rs.Name = "RunspaceX"
$Rs.Open()
$Ps.Runspace = $Rs

$Ps1Code = Get-Content -Path .\Skript.ps1 -Raw

[void]$Ps.AddScript($Ps1Code)

$Ps.Invoke()

$Rs.Close()
```

Damit das Skript durch den Aufruf von *$Ps.Invoke()* nicht einfach „durchrauscht“, wird in dieser Zeile ein Haltepunkt gesetzt. Damit hält das Skript in der ISE vor der Ausführung des Skripts in dem neuen Runspace an. Diese Technik ist aber nur dem konkreten Beispiel geschuldet. In der Praxis wird man situationsbedingt eine andere Herangehensweise wählen, die bewirkt, dass das zu debuggende Skript wartet.

Im nächsten Schritt wird in einer PowerShell-Konsole das *Enter-PSHost*-Cmdlet mit der Id des PowerShell_ISE-Prozesses ausgeführt:

```
Get-Process -Name "PowerShell_ISE" | Enter-PSHost
```

Der Prompt ändert sich in [Prozess: 1234], wobei 1234 stellvertretend für die Prozess-Id der ISE steht.

Der angelegte Runspace heißt „RunspaceX“. Achten Sie daher auf jene Runspaces, die den Namen „RunspaceX“ tragen und deren *State*-Eigenschaft den Wert „Open“ und deren *Availability*-Eigenschaft den Wert „Available“ besitzt. Für den nächsten Schritt wird Id dieses Runspaces benötigt.

Versetzen Sie durch das *Debug-Runspace*-Cmdlet jenen Runspace in den Debug-Modus, der über seine Id angegeben wird. Führen Sie das Skript in der ISE per [F5] weiter aus, so dass durch den Aufruf per *Invoke()* das Skript im Runspace „RunspaceX“ ausgeführt wird. Der Konsolenhost sollte dadurch in den Debug-Modus übergehen und die erste Zeile des Skripts anzeigen. Damit werden im Prompt des Konsolenhosts drei Angaben angezeigt: Der Debug-Modus, der Prozess und der Runspace. Über die bekannten Debug-Kommandos wird das Skript im RunspaceX unter der Kontrolle des Debuggers ausgeführt. Eine kleine Einschränkung besteht darin, dass alle Ausgaben des Skripts erst am Ende gesammelt ausgegeben werden.

Über [Strg]+[C] wird die Debug-Sitzung im Konsolenhost wieder beendet, per *exit* wird die Host-Anwendung wieder verlassen.

14.7 Zusammenfassung

Das Debuggen eines PowerShell-Skriptes ist eine wichtige Technik, um Fehler aufzuspüren und allgemein die Ausführungslogik eines Skripts besser nachvollziehen zu können. Das Debuggen kommt auch für PowerShell-Einsteiger in Frage. In größeren Skripten kommt es darauf an, dass Haltepunkte nicht immer aktiv werden, sondern in Abhängigkeit einer Bedingung. Außerdem kann es praktisch sein, wenn ein Skript, das remote ausgeführt wird, auch remote debuggt werden kann.

Sicherheit

15

Zusammenfassung

In diesem Kapitel geht es um alle jene Themen, die etwas mit dem Thema Sicherheit im Zusammenhang mit der PowerShell zu tun haben. Dazu gehören die „Klassiker" Ausführungsrichtlinien, signierte Skripte und Umgang mit Credentials und die modernen Themen Verschlüsseln von Zeichenketten und „Just Enough Administration" (JEA) für eingeschränkte PowerShell-Remoting-Endpunkte.

15.1 Die Rolle der Ausführungsrichtlinie

Die Ausführung eines Skriptes wird über eine Richtlinie gesteuert, die Ausführungsrichtlinie (engl. „execution policy"). Sie entscheidet darüber, ob auf einem Computer Ps1-Dateien ausgeführt werden oder nicht. Früher war die Voreinstellung „Restricted", so dass keine PowerShell-Skripte ausgeführt werden können. Ab Windows Server 2012 ist bei Windows Server die Voreinstellung mit „RemoteSigned" weniger restriktiv. Unter dieser Einstellung werden Skripte ausgeführt, sofern die Ps1-Datei keinen *Zone.Identifier*-Stream mit der Zonen-Id 3 enthält und nicht signiert ist.

Intern basiert die Ausführungsrichtlinie auf einem Eintrag in der Registry mit dem Namen „ExecutionPolicy": Für alle Benutzer unter *HKLM:\Software\Microsoft\PowerShell\1\ShellIds*, für den aktuellen Benutzer entsprechend unter *HKCU:\Software\Microsoft\PowerShell\1\ShellIds*. Daher kann die Ausführungsrichtlinie für alle Benutzer nur in einer Administrator-PowerShell geändert werden. Über den *Scope*-Parameter des *Set-ExecutionPolicy*-Cmdlets wird festgelegt, auf welcher Ebene die Einstellung wirksam werden soll. Die Einstellung gilt für alle Host-Anwendungen, allerdings gibt es für die 32- und die 64-Bit-Versionen eine eigene Einstellung. Das bedeutet konkret, dass wenn in einer 32-Bit-Hostanwendung die Richtlinie geändert wird, sich diese nicht auf

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_15

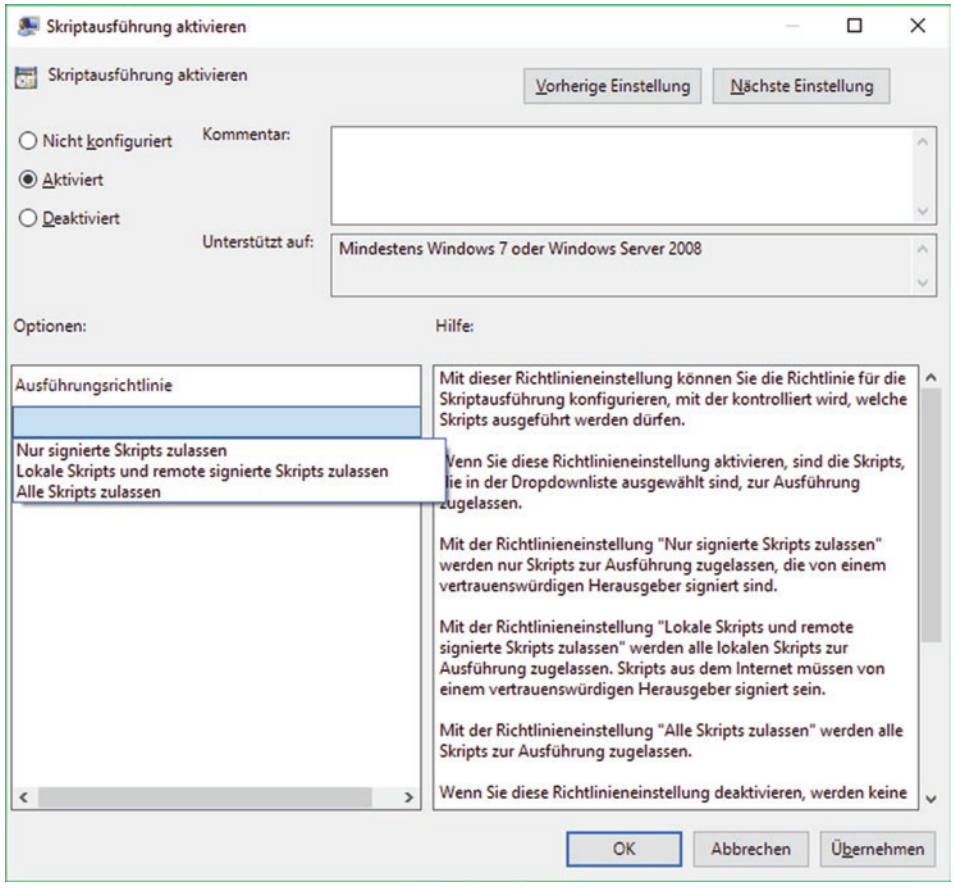

Abb. 15.1 Die Ausführungsrichtlinie wird in einer Domäne am einfachsten über eine Gruppenrichtlinie verteilt

die 64-Bit-Hostanwendungen auswirkt und umgekehrt. Eine Richtlinie muss daher für beide Plattformen gesetzt werden.

In der Praxis wird die Einstellung auf der Ebene eines Benutzerkontos oder des Computers über eine Gruppenrichtlinie durchgeführt (Abb. 15.1), so dass sie automatisch mit der nächsten Anmeldung wirksam wird.

15.2 Umgang mit Credentials

Bei PowerShell-Commands wird eine Authentifizierung in den meisten Fällen über ein Argument vom Typ *PSCredential* durchgeführt. Der Typ besitzt die Eigenschaften *Username* und *Password* und eine Methode mit dem unscheinbaren Namen *GetNetworkCredential*, über die sich das verschlüsselte Kennwort im Klartext abrufen lässt.

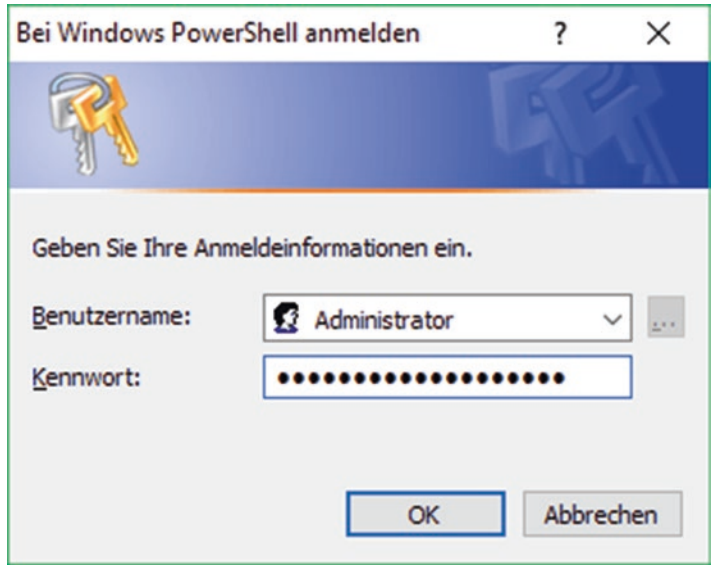

Abb. 15.2 Das Get-Credential-Cmdlet zeigt einen vertrauten Anmeldedialog an

Credentials bzw. ein *PSCredential*-Objekt erhält man am einfachsten per *Get-Credential*-Cmdlet (Abb. 15.2). Hier gibt es eine kleine Besonderheit zu beachten. Sobald der Parameter *Username* gesetzt wird, führt der folgende Aufruf dazu, dass auch für den *Message*-Parameter ein Wert übergeben werden muss:

```
$Cred = Get-Credential -Username Administrator
```

Lässt man den Parameternamen weg, geht es auch ohne *Message*-Parameterwert. Der Grund ist, dass in dieser Schreibweise der Name dem *Credential*-Parameter zugeordnet wird und es in diesem Parametersatz keinen *Message*-Parameter gibt.

Die *Password*-Eigenschaft eines *PSCredential*-Objekts ist vom Typ *SecureString*. Dahinter steht eine Klasse im Namespace *System.Security*, die nicht Teil der PowerShell, sondern der .NET-Laufzeit ist. Einen Secure String liefern die Cmdlets *ConvertTo-SecureString* und *Read-Host.* Ersteres konvertiert entweder eine verschlüsselte Zeichenkette oder eine „plain text"-Zeichenkette, letzteres nimmt eine Zeichenkette von der Tastatur entgegen und verschlüsselt sie bei gesetztem *AsSecure*-Parameter. Für die Ver- und Entschlüsselung wird derselbe Schlüssel (engl. „key") verwendet. Standardmäßig wird der Schlüssel vom System generiert. In den Schlüssel fließen die SID des Benutzerkontos und die des Computers ein. Das bedeutet, dass eine Zeichenkette so verschlüsselt wird, dass sie nur unter dem selben Benutzerkonto auf dem Computer, auf dem sie verschlüsselt wurde, wieder entschlüsselt werden kann. Wer einen Secure String portabel halten möchte, muss einen eigenen Key zur Verfügung stellen, im einfachsten Fall als eine Folge von Byte-Zahlen. Dieser Schlüssel muss beim Entschlüsseln der Zeichenkette auf einem anderen Computer natürlich vorhanden sein. Das bedeutet, dass der Schlüssel, in der Regel als Datei, zusätzlich gesichert werden

muss, da er nicht mehr durch das Betriebssystem verwaltet wird. Jeder, der den Schlüssel findet, kann eine damit verschlüsselte Zeichenkette entschlüsseln.

Beispiel

Das folgende Beispiel ist etwas umfangreicher. Es zeigt, wie ein Schlüssel erzeugt wird mit dem ein bereits als Secure String vorliegendes Kennwort in eine Zeichenkette konvertiert wird. Die verschlüsselte Zeichenfolge kann, zusammen mit dem Schlüssel, auf einen anderen Computer übertragen und dort mit einem Benutzernamen als *PSCredential*-Objekt für eine Authentifizierung verwendet werden.

```
<#
 .Synopsis
  Ein SecureString wird mit einem eigenen Schlüssel in eine Zeichenkette
konvertiert
#>

# Das Kennwort als SecureString holen
$PwSec = Read-Host -Prompt "Pw?" -AsSecureString

# Einen AES-Schlüssel holen
$AESKey = New-Object -TypeName "Byte[]" -ArgumentList 32
[Security.Cryptography.RNGCryptoServiceProvider]::Create().GetBytes($AESK
ey)

# Das Kennwort als SecureString mit dem Key in eine Zeichenkette
konvertieren
$Pw = ConvertFrom-SecureString -SecureString $PwSec -Key $AESKey

# Den Schlüssel in eine Datei speichern
$AESKeyFilePath = "AESKey.dat"
$PwPath = "Pw.dat"

$AESKey | Set-Content -Path $AESKeyFilePath

# Das Kennwort ebenfalls in eine Datei speichern
# Beide Dateien sollte per Zugriffsberechtigungen geschützt werden
$Pw | Set-Content -Path $PwPath

# Später wird das Kennwort wieder verwendet
$Pw = Get-Content -Path $PwPath

$AESKey = Get-Content -Path $AESKeyFilePath
$PwSecNeu = ConvertTo-SecureString -String $Pw -Key $AESKey

$Cred = [PSCredential]::new("Administrator", $PwSecNeu)

# Credentials speichern
$Cred | Export-Clixml -Path .\Cred.xml
```

15.2.1 Einen Secure String lesbar machen

Ein Secure String ist nicht ganz so sicher wie der Name es impliziert. Er lässt sich über die *GetNetworkCredential()*-Methode des *PSCredential*-Objekts und dessen *Password*-Eigenschaft sehr einfach wieder in eine „plain text"-Zeichenkette entschlüsseln. Da dies

aber nur dem Benutzer, der die Zeichenkette verschlüsselt hat, auf dem Computer möglich ist, auf dem sie verschlüsselt wurde, ist dies keine Sicherheitslücke, sondern eher eine Komforteinrichtung (man muss die Dinge ja positiv sehen).

Beispiel

Der folgende Aufruf gibt einen Secure String in lesbarer Form aus.

```
$Cred.GetNetworkCredential().Password
```

15.2.2 Einen Secure String anlegen

Einen Secure String erhält man bei der PowerShell entweder über das *ConvertTo-SecureString*-Cmdlet bei gesetzten *AsPlainText*- und *Force*-Parametern oder über das *Read-Host*-Cmdlet bei gesetztem *AsSecureString*-Parameter. Es gibt eine dritte Alternative, die auch unabhängig von der PowerShell funktioniert. Sie besteht darin, ein Objekt vom Typ *SecureString* anzulegen und über die *AppendChar()*-Methode jedes Zeichen einzeln hinzuzufügen (auch das ist gewollt, damit die unsichere Methode, einen Secure String zu erhalten nicht zu bequem wird).

Beispiel

Der folgende Befehl legt ein *SecureString*-Objekt direkt an.

```
<#
 .Synopsis
 Direktes Anlegen eines Secure Strings
#>
$PwSec = New-Object -TypeName SecureString
"geheim".ToCharArray().ForEach{$PwSec.AppendChar($_)}
```

15.2.3 Secure String-Dateien sicher speichern

Per *ConvertFrom-SecureString*- oder *Export-CliXml*-Cmdlet lässt sich ein Secure String im Textformat in einer Datei speichern. Die Frage ist nur, wo die Datei im Windows-Dateisystem am besten aufgehoben ist. Die Antwort: Im einfachsten Fall in einem Verzeichnis des Benutzerprofils unter *%AppPath%*, das bezüglich seiner Zugriffsberechtigungen eingeschränkt ist, so dass nur ein bestimmtes Benutzerkonto Lese- und Schreibberechtigungen besitzt.

▶ **Hinweis** Geht es um das Verschlüsseln von Zeichenketten, ist das mit der Version 5.0 eingeführte *Protect-CmsMessage*-Cmdlet dafür besser geeignet, da es für die Verschlüsselung ein Zertifikat verwendet.

15.3 Umgang mit Zertifikaten

Ein Zertifikat ist eine binäre Datenstruktur im X.509-Format. Die PowerShell bietet keine spezifischen Cmdlets für den Umgang mit Zertifikaten, sondern spricht Zertifikate der lokalen Zertifikatablage über ein Laufwerk mit dem Namen „Cert“ an. Die Cmdlets *Get-Item* und *Get-ChildItem* holen von diesem Laufwerk Objekte vom Typ *X509Certificate2* (aus dem Namespace *System.Security.Cryptography.X509Certificates*).

Da ein Zertifikat keinen Namen oder Pfad besitzt, muss es über eine andere Eigenschaft eindeutig identifiziert werden. Das direkte Pendant zum Namen ist der „Daumenabdruck“ oder „Fingerabdruck“ (engl. „thumbprint“).

▶ **Hinweis** Der Thumbprint eines Zertifikats ist eine 40-stellige Hexadezimalzahl (neben Ziffern kommen auch die Buchstaben A bis F vor). Dank Tab-Vervollständigung ist die Eingabe in der PowerShell-Konsole aber kein Problem.

Beispiel

Der folgende Befehl holt per *Get-Item*-Cmdlet ein Zertifikat anhand seines Thumbprints.

```
$Cert = Get-Item –Path
Cert:\LocalMachine\My\2C47D563F5734230DBB3B9D1E80BEAFDEA720E50
```

▶ **Hinweis** Soll ein Zertifikat über eine seiner zahlreichen anderen Eigenschaften geholt werden, geschieht dies über das *Where-Object*-Cmdlet. Sie werden feststellen, dass das per *Get-Item* geholte Objekt eine Reihe von „Spezialeigenschaften“ besitzt.

Beispiel

Der folgende Befehl holt ein Zertifikat anhand seines „Common Names“ (CN).

```
Get-ChildItem Cert:\CurrentUser\My\ | Where-Object Subject -eq
"CN=PsKurs"
```

Die Rückgabe ist stets ein Objekt vom Typ *X509Certificate2*.

▶ **Tipp** Dies ist ein nützlicher Tipp: Um ein Zertifikat in der Zertifikatsverwaltungskonsole anzuzeigen, genügt es, dem Pfad ein *Invoke-Item*-Cmdlet voranzustellen.

Beispiel

Der folgende Befehl zeigt ein einzelnes Zertifikat in der MMC-Verwaltungskonsole an (das das Zertifikat auch ausgewählt wird, scheint nur auf einem englischsprachigen Windows zu funktionieren, da die Ablagen in einer lokalisierten Version andere Namen tragen, z. B. „Eigene Zertifikate“ anstelle von „My“).

```
Get-Item –Path
Cert:\LocalMachine\My\2C47D563F5734230DBB3B9D1E80BEAFDEA720E50 | Invoke-
Item
```

15.4 Weiterer Umgang mit Zertifikaten

Bereits mit der Version 3.0 wurde der *Certificate*-Provider, der das *Cert*-Laufwerk zur Verfügung stellt, erweitert. Seitdem sind auch Operationen wie das Löschen von Zertifikaten per *Remove-Item*-Cmdlet und das Anlegen neuer Ablagen (aber nicht von Zertifikaten) per *New-Item*-Cmdlet möglich. Trotz kleiner Verbesserungen ist der Provider noch limitiert. Ein Kopieren von Zertifikaten per *Copy-Item* ist leider nicht möglich, ein Verschieben von Zertifikaten per *Move-Item* nur innerhalb derselben Ablage („CurrentUser“ und „LocalMachine“). Die ItemProperty-Cmdlets können mit Zertifikaten generell nicht verwendet werden. Möchte man ein Zertifikat auf einen anderen Computer übertragen, geschieht dies per Export in eine Textdatei, die auf dem Zielcomputer wieder importiert wird. Entweder über die Cmdlets *Export-Certificate* und *Import-Certificate* aus dem *PKI*-Modul oder, sollte das Modul nicht zur Verfügung stehen, über das Befehlszeilentool *Certutil.exe*.

- **Hinweis** In der Hilfe werden die Möglichkeiten des Certificate-Providers über „help certificate“ angezeigt.

- **Hinweis** Vorsicht: Der Umgang mit Zertifikaten mit Standardkommandos wie etwa *del* birgt auch gewisse Risiken. Ein „del *“ löscht zum Beispiel sämtliche Zertifikate in der aktuell voreingestellten Zertifikatablage.

15.4.1 Auflisten bestimmter Zertifikate

Im Allgemeinen werden Zertifikate zweckgebunden eingesetzt. Es gibt Zertifikate für die Server-Authentifizierung, für die Client-Authentifizierung, für die Signierung von Skripten, für das Verschlüsseln von Dokumenten usw. Die erweiterten Möglichkeiten eines Zertifikats werden über die Eigenschaft *Eku* („Enhanced Key Usage“) festgelegt. Diese Eigenschaft muss aber nicht direkt abgefragt werden, da der Certificate-Provider dynamische Parameter wie *CodeSigningCert* oder *SSLServerAuthentication* zur Verfügung stellt, über die nur Zertifikate des angegebenen Typs zurückgegeben werden.

Beispiel

Der folgende Befehl gibt alle Zertifikate auf dem lokalen Computer zurück, die für eine SSL-Serverauthentifizierung geeignet sind.

```
dir cert:\localmachine -Recurse -SSLServerAuthentication
```

15.5 Zertifikate anlegen

Das Anlegen eines Zertifikats per New-Item-Cmdlet ist nicht möglich. Dazu müssten zum einen zu viele dynamische Parameter zur Verfügung gestellt werden, zum anderen wird ein „echtes“ Zertifikat immer von einer „Certificate Authority“ (CA) bezogen. Eine solche

CA kann bei Windows Server über die Active Directory Zertifikatdienste als Feature hinzugefügt werden. Im Internet gibt es Schritt für Schritt-Anleitungen für das Bereitstellen einer Root CA, dem obersten Zertifikat in der baumartigen Hierarchie, mit dem andere Zertifikate signiert und damit „ausgestellt" werden. Der ganze Ablauf ist bei Windows Server allerdings komplizierter als es den Anschein haben könnte, so dass eine funktionierende PKI („Private Key Infrastructure") in Unternehmen und Organisationen nach wie vor nicht selbstverständlich ist.

15.5.1 Selbstsignierte Zertifikate erstellen

Ohne das Vorhandensein eines von einer Root CA ausgestelltes Stammzertifikat muss ein selbstsigniertes Zertifikat ausgestellt werden. Der Wert eine solchen Zertifikats besteht in erster Linie darin, dass es ein Zertifikat ist und damit eine Unverwechselbarkeit gegeben ist, die zum Beispiel für das Signieren von Skripten oder das Verschlüsseln einer HTTPS-Verbindung ausreichend ist. Wird ein selbstsigniertes Zertifikat in die Ablage der vertrauenswürdigen Zertifizierungsstellen kopiert, wird dem Herausgeber des Zertifikats zumindest auf dem lokalen Computer vertraut. Wie selbstsignierte Zertifikate im Rahmen der PowerShell erstellt werden, wird in diesem Abschnitt gezeigt.

Unter Windows Server 2008 R2 und Windows 7 gibt es kein Cmdlet für das Ausstellen eines selbstsignierten Zertifikats. Das in der PowerShell-Hilfe an einigen Stellen verwendete Tool *Makecert.exe* ist offiziell nicht mehr verfügbar, so dass es heutzutage nicht mehr verwendet werden sollte. Ohne dieses Tool müsste ein Zertifikat per PowerShell zu Fuß mit Hilfe der Klasse *X509Certificate2* aus dem Namespace *System.Security.Cryptography.X509Certificates* angelegt werden. Die Mühe muss sich allerdings niemand machen, denn es gibt ein sehr gutes Skript, das sich als inoffizieller Nachfolger von *Makecert.exe* versteht. Das Skript heißt *New-SelfSignedCertificateEx.ps1* und stammt von dem Sicherheitsexperten *Vadim Podans*, der unter anderem Autor eines PowerShell-Moduls zum Thema PKI ist. Mit diesem Skript wird das Erstellen von selbstsignierten Zertifikaten für alle denkbaren Einsatzzwecke sehr einfach. Das Skript steht unter anderem im Rahmen der TechNet Script Gallery zur Verfügung.

- **Tipp** Wer aus irgendwelchen Gründen *Makecert* einsetzen möchte, findet unter der folgenden Adresse einen Blog-Eintrag, der den Umgang mit dem Tool sehr schön beschreibt: http://www.it-netzwerk-systeme.com/zertifikate-selbst-erstellen.

- **Hinweis** Die allgemeine Rolle des Zertifikats (genauer gesagt seines öffentlichen Schlüssels) wird über den Parameter *KeyUsage* festgelegt. Wichtiger ist der Parameter *EnhancedKeyUsage*, über den der Verwendungszweck des Zertifikats im Detail festgelegt wird. Auf diesen Parameter kommt es zum Beispiel an, wenn ein Zertifikat für die Codesignierung benötigt wird. Der Parameterwert wird

entweder als Bezeichnung (in der jeweiligen Landessprache, zum Beispiel „Codesignatur" für die Codesignierung) oder direkt über den OID-Wert (Object Identifier) festgelegt. Die Tabelle stellt einige OID-Werte zusammen, alle anderen muss man im Internet recherchieren.

Zertifikatrolle	OID-Wert
Serverauthentifizierung	1.3.6.1.5.5.7.3.1
Clientauthentifizierung	1.3.6.1.5.5.7.3.2
Code Signing	1.3.6.1.5.5.7.3.3

▶ **Tipp** Es ist nicht immer einfach, eine „Key Usage-Bezeichnung" in einer bestimmten Landessprache herauszufinden. Für „Code Signing" muss bei einem Windows mit deutscher Sprache zum Beispiel „Codesignatur" verwendet werden. Am einfachsten ist es, im Eigenschaftendialog eines vorhandenen Zertifikats im Zertifikatmanager (*Certmgr.msc*) nachzuschauen, sofern ein passendes Zertifikat vorhanden ist (Abb. 15.3).

Ab Windows Server 2012 und Windows 8 gibt es das praktische *New-SelfSignedCertificate*-Cmdlet. Es dient dem Anlegen von selbstsignierten Zertifikaten, also ohne Mitwirkung einer CA, für das Absichern einer HTTPS-Verbindung. Ein für die Codesignierung und damit für die Signierung von PowerShell-Skripten geeignetes Zertifikat kann damit nicht erstellt werden. Das ist mit dem Cmdlet erst ab Windows Server 2016 und Windows 10 AE möglich. Ab diesen Windows-Versionen wurde das Cmdlet um zahlreiche Parameter erweitert, so dass sich alle Zertifikatstypen anlegen lassen.

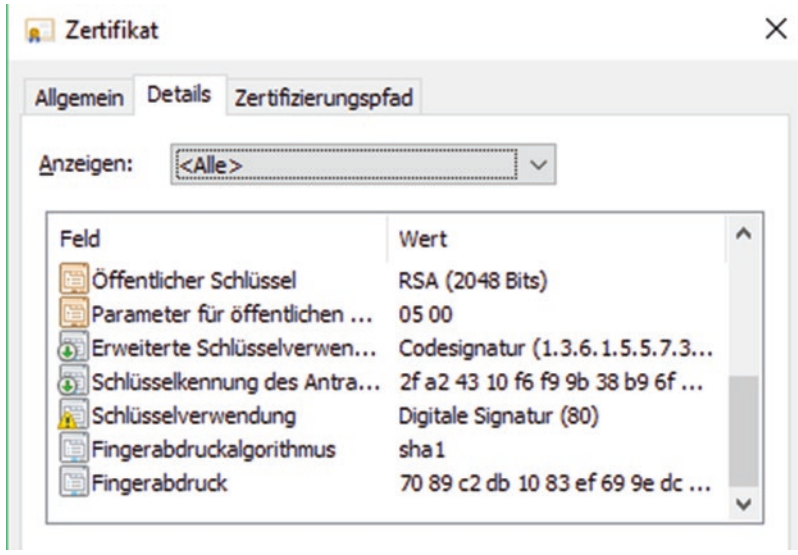

Abb. 15.3 Die Eigenschaften eines Zertifikats

Beispiel

Der folgende Befehl ist für die Ausführung unter Windows Server 2008 R2 und Windows 7 gedacht. Er legt mit Hilfe des Skripts *New-SelfSignedCertificateEx.ps1* ein selbstsigniertes Zertifikat an, das für die Codesignierung und damit auch für die Signierung eines PowerShell-Skripts geeignet ist:

```
New-SelfSignedCertificateEx -Subject "CN=PSKurs" -StoreLocation
CurrentUser -EnhancedKeyUsage "Codesignatur"
```

Damit liegt ein für die Codesignierung geeignetes Skript in der Ablage *CurrentUser\My* vor. Damit das Skript für die Signierung per *Set-AuthenticodeSignature*-Cmdlet verwendet werden kann, muss das Zertifikat in die Ablage der vertrauenswürdigen Stammzertifizierungsstellen („Root“) kopiert werden. Ansonsten ist der Zertifikatstatus des Skriptes nicht gültig. Per *Copy-Item* ist dies leider nicht möglich. Ab Windows Server 2016 und Windows 10 AE gibt es dafür im PKI-Modul die Cmdlets *Export-Certificate* und *Import-Certificate*. Am einfachsten geht es im Zertifikatmanager (*Certmgr.msc*) per Drag & Drop. Das Hinzufügen muss jedes Mal explizit bestätigt werden.

► **Tipp** Wer das Kopieren eines Zertifikats unbedingt per PowerShell ohne den Umweg über einen Export/Import durchführen möchte, kann dies natürlich tun. Zum Beispiel mit einer kleinen Function, in der die Ablagen über Klassen der .NET-Laufzeit direkt angesprochen werden. Auch bei diesem Weg muss das Hinzufügen explizit bestätigt werden.

Beispiel

Die Function *Copy-Cert* kopiert ein Zertifikat von einer Ablage in eine andere Ablage.

```
function Copy-Cert
{

param([System.Security.Cryptography.X509Certificates.X509Certificate2]$Ce
rt, [String]$StoreScopeSource, [String]$StoreNameSource,
[String]$StoreScopeDest, [String]$StoreNameDest)

  $SourceStore = Get-Item -Path
"Cert:\$StoreScopeSource\$StoreNameSource"
  $DestStore = Get-Item -Path "Cert:\$StoreScopeDest\$StoreNameDest"
  $DestStore.Open("ReadWrite")
  $DestStore.Add($Cert)
  $DestStore.Close()}
```

► **Hinweis** Ab Windows Server 2012 und Windows 8 gibt es im Modul PKI 17 Cmdlets für den Umgang mit Zertifikaten.

Beispiel

Das folgende Beispiel verwendet das erweiterte *New-SelfSignedCertificate*-Cmdlet um ein Zertifikat anzulegen, das für Codesignierung geeignet ist.

```
New-SelfSignedCertificate -Subject "CN=Win10Mobil" -CertStoreLocation
"Cert:\CurrentUser\My" -Type CodeSigningCert  -KeyUsageProperty Sign
```

Damit das Zertifikat für die Signierung eines PowerShell-Skriptes verwendet werden kann, muss es in die Root-Ablage („Vertrauenswürdige Stammzertifikate)“ kopiert werden. Dazu wird es zuerst mit dem *Export-Certificate*-Cmdlet in eine Cer-Datei exportiert, die anschließend mit dem *Import-Certificate*-Cmdlet wieder importiert wird. Der Thumbprint des Zertifikats wird zuerst einer Variablen zugewiesen.

```
$Cert = Get-Item -Path Cert:\CurrentUser\My\$Thumb
Export-Certificate -Type CERT -FilePath C:\ScriptSignCert.Cer -Cert $Cert

Import-Certificate -FilePath "C:\ScriptSignCert.Cer" -CertStoreLocation
"Cert:\LocalMachine\Root" -Verbose
```

▶ **Hinweis** Ein universelles Modul für den Umgang mit Zertifikaten und einer PKI stammt von Sicherheitsexperten *Vadim* Podans, der sich auf die PowerShell spezialisiert hat. Download unter https://pspki.codeplex.com.

15.6 Skripte signieren

PowerShell-Skripte können signiert werden. Das bietet zwei kleinere Vorteile: Das Skript kann einer Person oder Institution zugeordnet werden. Das Skript besitzt eine Prüfsumme, so dass sich feststellen lässt, ob das Skript nach der Signierung noch einmal verändert wurde. Für die Signierung ist das Cmdlet *Set-AuthenticodeSignature* zuständig. Es erwartet den Pfad der Ps1-Datei und ein Zertifikat, das für die Codesignierung geeignet ist.

Beispiel

Der folgende Befehl signiert die Skriptdatei *Test.ps1* mit einem zuvor angelegten Zertifikat mit dem Subject „CN=PsKurs“.

```
$Cert = (dir Cert:\CurrentUser -Recurse -CodeSigningCert | Where Subject
-eq "CN=PsKurs")[0]
Set-AuthenticodeSignature -Certificate $Cert -FilePath $Ps1Pfad  -Force -
Verbose
```

Sollen nur noch signierte Skripte ausgeführt werden, muss die Ausführungsrichtlinie auf „AllSigned“ gesetzt werden.

Beispiel

Der folgende Befehl setzt die Ausführungsrichtlinie für den aktuellen Benutzer auf „AllSigned“.

```
Set-ExecutionPolicy -ExecutionPolicy "AllSigned" -Scope CurrentUser
```

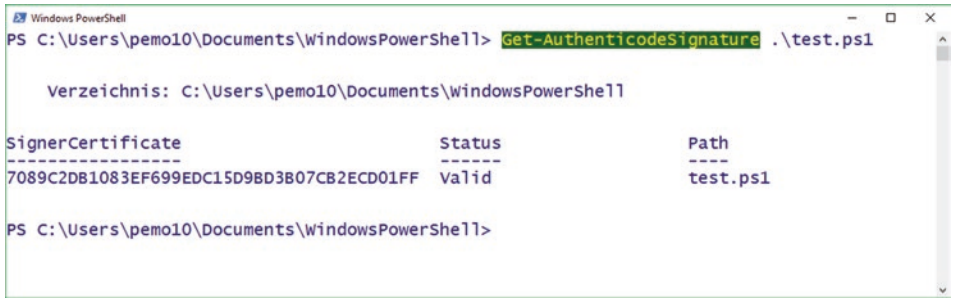

Abb. 15.4 Das Skript .\Test.ps1 wurde mit einem gültigen Zertifikat signiert

Wird ein signiertes Skript ausgeführt, dessen Herausgeberzertifikat nicht von einer Root-CA signiert wurde, muss die Ausführung des Skripts trotz Signierung bestätigt werden. Es werden folgende Optionen angeboten: „Noch nie ausgeführt“, „Nicht ausführen“, „Einmal ausführen“ und „Immer ausführen“. Per ? werden alle Optionen beschrieben. Bei „Immer ausführen“ wird das Zertifikat in die Ablage „Vertrauenswürdige Herausgeber“ kopiert, bei „Noch nie ausgeführt“ (besser „Niemals ausführen“) entsprechend in die Ablage „Nicht vertrauenswürdige Zertifikate/Zertifikate“.

Wird an einem signierten Skript eine Änderung vorgenommen, muss das Skript erneut signiert werden. Ansonsten kann es bei „AllSigned“ nicht mehr ausgeführt werden. Per *Get-AuthenticodeSignature*-Cmdlet stellt man fest, ob das Skript mit einem gültigen Zertifikat signiert wurde (Abb. 15.4).

15.7 Zeichenketten verschlüsseln

Für das Verschlüsseln von allgemeinen Zeichenketten ist ein Secure String nicht gut geeignet. Flexibler sind zwei Alternativen: Die Verwendung einer externen Funktionsbibliothek wie z. B. *CipherNet Lite* oder das mit der Version 5.0 eingeführte *Protect-CmsMessage-Cmdlet*.

15.7.1 Texte verschlüsseln und entschlüsseln mit CipherNet

CipherNet ist der Name einer kleinen, aber für viele Zwecke vollkommen ausreichenden Funktionsbibliothek für das Ver- und Entschlüsseln von Texten. In der Version *CipherNet Lite* ist das Produkt kostenlos. Es gibt natürlich viele andere solcher Funktionsbibliotheken auf der Basis des .NET Frameworks, für *CipherNet* spricht, das sie relativ unkompliziert ist. Eine sehr attraktive Alternative, die in diesem Buch aber nicht behandelt wird, ist *Bouncy Castle* (Download unter http://bouncycastle.org/csharp/index.html). Die Downloadadresse von *Cipher Lite* ist http://www.obviex.com/CipherLite/Downloads.aspx. Der Download umfasst mehrere Dateien, es kommt auf die Datei *CipherLite.dll* an, die per *Add-Type*-Cmdlet geladen wird.

Beispiel

Das folgende Beispiel verschlüsselt eine Zeichenkette mit Hilfe von *Cypher.Net Lite* und entschlüsselt sie wieder.

```
$DllPath = "C:\Program Files\CipherLite.NET\CipherLite.dll"

Add-Type -Path $DllPath

$CyPw = "abcdefg"
$Pw = "geheim+123"
$Cy = New-Object -TypeName Obviex.CipherLite.Crypto -ArgumentList $CyPw
$PwSec = $Cy.Encrypt($pw)
$Cy.Decrypt($PwSec)
```

15.7.2 Zeichenketten mit Zertifikaten verschlüsseln

Bis zur Version 5.0 war der Umgang mit verschlüsselten Zeichenketten mit den Bordmitteln der PowerShell nicht möglich. Dank des *Proctect-CmsMessage*-Cmdlets wird das Verschlüsseln von Zeichenketten auf der Grundlage eines Zertifikats einfach und vor allem sicher. Im ersten Schritt muss das Zertifikat angelegt werden, was mit Hilfe einer Vorlagentextdatei, die alle Eigenschaften des Zertifikats enthält, sehr einfach ist. Dabei kommt es auf den Namen an, der über die Einstellung „Subject“ festgelegt wird, denn über diesen Namen wird beim Aufruf von *Protect-CmsMessage* das Zertifikat am einfachsten ausgewählt.

▶ **Hinweis** Es ist wichtig, dass die Zertifikatvorlagendatei im ANSI-Format gespeichert wird. Speichern mit der ISE kommt daher nicht in Frage.

```
; muss im ANSI-Format gespeichert werden
[Version]
Signature = "$Windows NT$"

[Strings]
szOID_ENHANCED_KEY_USAGE = "2.5.29.37"
szOID_DOCUMENT_ENCRYPTION = "1.3.6.1.4.1.311.80.1"

[NewRequest]
Subject = "CN=Pskurs"
MachineKeySet = false
KeyLength = 2048
KeySpec = AT_KEYEXCHANGE
HashAlgorithm = Sha1
Exportable = true
RequestType = Cert
KeyUsage = "CERT_KEY_ENCIPHERMENT_KEY_USAGE |
CERT_DATA_ENCIPHERMENT_KEY_USAGE"
ValidityPeriod = "Years"
ValidityPeriodUnits = "1000"

[Extensions]
 %szOID_ENHANCED_KEY_USAGE% = "{text}%szOID_DOCUMENT_ENCRYPTION%"
```

Beispiel

Für das folgende Beispiel wird davon ausgegangen, dass sich die Zertifikatdefinition in der Datei „PsKursZertifikat.inf“ befindet, die zuvor mit einem Editor angelegt wurde. Über den Aufruf des Kommandozeilentools *Certreq.exe* wird aus der Datei ein Zertifikat, das automatisch in die Zertifikatablage eingefügt wird.

```
Certreq -new PsKursZertifikat.inf PsKursZertifikat.cer
```

Beispiel

Der folgende Befehl listet alle Zertifikate auf, die für eine Dokumentverschlüsselung geeignet und dem aktuellen Benutzer zugeordnet sind.

```
dir -Path Cert:\CurrentUser\My –DocumentEncryptionCert
```

Damit liegt das Zertifikat vor, so dass damit per *Proctect-CmsMessage*-Cmdlet Texte verschlüsselt werden können.

Beispiel

Der folgende Befehl verschlüsselt einen Text per *Proctect-CmsMessage*-Cmdlet unter Angabe eines Zertifikats und schreibt ihn in eine Datei.

```
$Pw = "Dies ist eine streng geheime Botschaft"
$Pw | Protect-CmsMessage -To "Cn=PsKurs" > Pw.dat
```

Über das *Unprotect-CmsMessage*-Cmdlet wird der Text wieder lesbar gemacht.

Beispiel

Der folgende Befehl entschlüsselt einen Text per *Unproctect-CmsMessage*-Cmdlet, der sich in einer Datei befindet.

```
Get-Content –Path Pw.dat | Unprotect-CmsMessage
```

Das *Get-CmsMessage*-Cmdlet liefert alle Details über die verschlüsselte Zeichenkette.

15.8 Umgang mit Zugriffsberechtigungen

Zugriffsberechtigungen spielen beim Zugriff auf unterschiedliche „Objekt-Typen“ unter Windows eine Rolle, nicht nur auf Dateien und Verzeichnisse, sondern zum Beispiel auch auf Registry-Schlüssel oder Active Directory-Objekte. PowerShell bietet für den Umgang mit Zugriffsberechtigungen die Cmdlets *Get-ACL* und *Set-ACL*. Das *Get-ACL*-Cmdlet holt die *Access Control List* für einen Gegenstand wie eine Datei, das *Set-ACL*-Cmdlet ordnet

einem Objekt eine neue ACL zu. Beide Cmdlets können nicht nur mit Dateien und Verzeichnissen verwendet werden, sondern generell mit allen Sorten von „Items“, die von einem PSProvider zur Verfügung gestellt werden. Dazu gehören neben Registry-Schlüsseln auch Active Directory-Benutzerkonten.

Beispiel

Der folgende Befehl geht von einem Verzeichnis *C:\ACLTest* aus. Er gibt alle Zugriffsberechtigungen für dieses Verzeichnis aus. Das Umschalten auf Listenformatierung ist erforderlich, damit alle Zugriffsberechtigungen auch sichtbar werden.

```
Get-ACL –Path C:\ACLTest | Format-List
```

Um Zugriffsberechtigungen entfernen und hinzufügen zu können, muss man ein wenig über die Hintergründe Bescheid wissen. Jedes Verzeichnis und jede Datei beziehungsweise allgemein jedes Objekt, für das Zugriffsberechtigungen existieren, besitzt eine Liste von Zugriffsberechtigungen. Wie es der Name dezent andeutet, handelt es sich um eine Liste von „Wer darf was?“-Berechtigungen. Diese Liste heißt *Access Control List*, kurz ACL. Jeder einzelne Eintrag heißt *Access Control Entry*, kurz ACE. Von den Eigenschaften eines ACE sind zwei Eigenschaften wichtig: Die Identität und die Berechtigungen. Die Identität steht für ein Benutzerkonto, die Berechtigung entsprechend für die Art des Zugriffs, die dem Benutzerkonto erlaubt wird. Geht es um ein Verzeichnis oder eine Datei, basiert das ACE-Objekt auf der Klasse *FileSystemAccessRule* im Namespace *System.Security.AccessControl*. Die beiden wichtigsten Eigenschaften sind entsprechend *IdentityReference* und *FileSystemRights*. Andere Eigenschaften geben an, ob die Berechtigungen vererbt wurden und an untergeordnete Objekte weitergegeben wurden.

Leider hatten sich die Entwickler der PowerShell dazu entschieden, Objekte zu verwenden, die direkt auf den Klassen der .NET-Laufzeit basieren anstatt diese Objekt etwas anwendungsfreundlicher zu gestalten. Dadurch wirkt der Umgang mit den ACLs etwas sperrig und wirklichkeitsfern im Vergleich zu Admintools wie *ICacls.exe*. Module wie *NTFSSecurity* von *Raimund Andrée* gleichen dies wieder aus, indem sie Cmdlets zur Verfügung stellen, mit deren Hilfe der Umgang mit ACLs und ACEs deutlich einfacher wird.

Beispiel

Das folgende Beispiel setzt auf das *NTFSSecurity*-Modul, das zuvor per *Install-Module* von der *PSGallery* hinzugefügt werden muss. Das *Get-NTFSAccess*-Cmdlet gibt die Zugriffsberechtigungen für ein Verzeichnis etwas übersichtlicher aus als es bei *Get-ACL* der Fall ist.

```
Get-NTFSAccess -Path C:\ACLTest
\
```

15.8.1 Entfernen von Zugriffsberechtigungen

Um von einem Verzeichnis oder einer Datei Zugriffsberechtigungen entfernen zu können, muss zuerst die Vererbung deaktiviert werden, so dass die vererbten Zugriffsberechtigungen in direkte Zugriffsberechtigungen umgewandelt werden. Im nächsten Schritt werden einzelne oder alle Zugriffsberechtigungen der Reihe nach durch einen Aufruf der *RemoveAccessRule*-Methode des ACL-Objekts entfernt. Sowohl nach dem Umwandeln der Vererbung als auch nach dem Entfernen aller Zugriffsregeln muss das Objekt per *Set-ACL* aktualisiert werden.

Beispiel

Das folgende Beispiel ist etwas umfangreicher. Es besteht aus einer Function, die von einem Verzeichnis, dessen Pfad beim Aufruf übergeben wird, alle Zugriffsberechtigungen entfernt und eine FullControl-Berechtigung für das Administratorkonto hinzufügt. Zuvor wird die Vererbung der Zugriffsberechtigungen für das Verzeichnis deaktiviert und alle bislang geerbten Zugriffsberechtigungen werden in „echte“ Zugriffsberechtigungen umgewandelt, so dass sie entfernt werden können.

```
<#
.Synopsis
 Entfernen aller vorhandenen Verzeichnisberechtigungen und Hinzufügen
einer neuen Berechtigung

function Clear-ACL
{
  [CmdletBinding()]
  param([String]$Pfad)
  $ACL = Get-Acl -Path $Pfad

  # Vererbung und Schutz außer Kraft setzen
  $ACL.SetAccessRuleProtection($true, $true)
  Set-ACL -Path $Pfad -AclObject $ACL

  # Wichtig: Get-ACL muss nach dem Aktualisieren erneut ausgeführt werden
  $ACL = Get-Acl -Path $Pfad
  # Alle Regeln durchgehen
  foreach($Rule in $ACL.Access)
  {
    Write-Verbose ("Identität: {0} Berechtigung: {1}" -f
$Rule.IdentityReference, $Rule.FileSystemRights)
    # Alle Zugriffsberechtigungen mit der SID der Zugriffsregel $Rule
entfernen
    $ACL.RemoveAccessRule($Rule) | Out-Null
  }

  # ACL aktualisieren
  Set-ACL -Path $Pfad -AclObject $ACL
  # Eine neue Regel hinzufügen
  $Rule = New-Object -TypeName
System.Security.AccessControl.FileSystemAccessRule -ArgumentList
"Administrator", "FullControl", "None", "None", "Allow"
  $ACL.AddAccessRule($Rule)

  # ACL aktualisieren
  Set-ACL -Path $Pfad -AclObject $ACL
}
```

15.9 Die Befehlsausführung protokollieren

Die Protokollierung der Befehlsausführung per *Start-Transcript*-Cmdlet war beim Konsolenhost seit der Version 1.0 möglich, bei der ISE leider erst mit der Version 5.0. Bereits mit einem Update (dem November-Update 2014) zur Version 4.0 wurden die Möglichkeiten zur Protokollierung deutlich erweitert. Es gibt zwei wesentliche Neuerungen: 1. Der Umfang der Protokollierung kann über Gruppenrichtlinien gesteuert werden. 2. Die Protokollierung bezieht sich nicht mehr ausschließlich auf die Aktivitäten in einer Host-Anwendung, sondern auf Wunsch auch auf einen Scriptblock, der Teil einer beliebigen Host-Anwendung sein kann. Die Ausführung des Scriptblocks hat einen Eintrag im Ereignisprotokoll zur Folge. Das Ziel ist es, dass bei Bedarf die Ausführung jedes PowerShell-Befehls auf einem Computer nachvollzogen werden kann. Da dies aber zu einer Verlangsamung der Ausführung führen kann, sollte diese Maßnahme nur punktuell und nicht pauschal eingesetzt werden.

15.9.1 Befehlsprotokollierung per Gruppenrichtlinien steuern

Die Protokollierung auf der Ebene eines Scriptblocks muss aktiviert werden, entweder per Gruppenrichtlinie ober über die Registry (Abb. 15.5). Die Gruppenrichtlinie ist **Computerkonfiguration -> Administrative Vorlagen -> Windows-Komponenten -> Windows PowerShell -> Protokollierung von PowerShell-Scriptblöcken aktivieren**. Bei der Aktivierung der Richtlinie kann angegeben werden, ob auch Start- und Stopereignisse protokolliert werden.

Ohne GPO-Editor kann die Scriptblockprotokollierung auch direkt in der Registry aktiviert werden. Der zuständige Schlüssel ist *HKey_Local_Machine\Software\Policies\Microsoft\Windows\PowerShell*. Der Unterschlüssel *ScriptBlockLogging* muss zuerst angelegt werden. Der zuständige Eintrag heißt *EnableScriptBlockLogging* und muss den Wert 1 erhalten, damit das Logging aktiviert wird.

Beispiel

Der folgende Befehl aktiviert die Scriptbock-Protokollierung für alle Benutzer.

```
$RegPfad =
"HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"
Set-ItemProperty -Path $RegPfad -Name EnableScriptBlockLogging -Value "1"
-Type DWORD
```

Wird anschließend ein Scriptblock ausgeführt, hat dies einen Eintrag im PowerShell-Ereignisprotokoll *Microsoft-Windows-PowerShell/Operational* mit der ID 4104 zur Folge. Jeder Scriptblock erhält im Eventlog-Eintrag eine eigene GUID. Informationen über die ausgeführten Befehle enthält die *Message*-Eigenschaft, die am besten per *Format-List* ausgegeben wird.

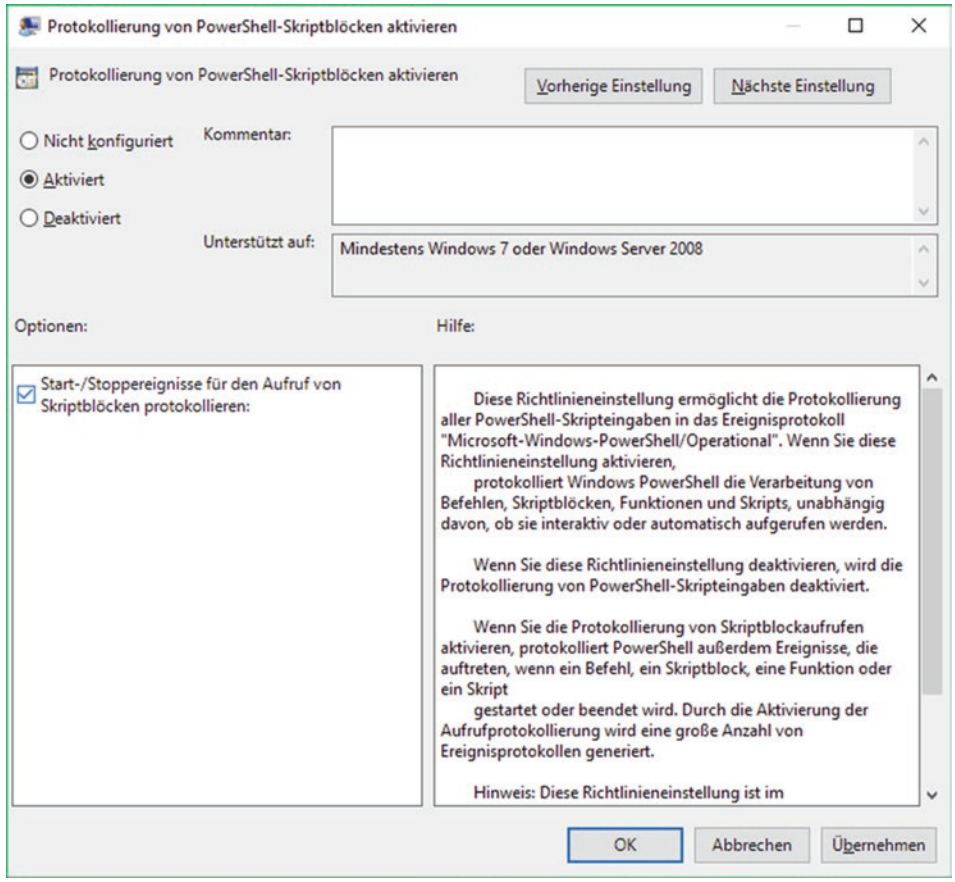

Abb. 15.5 Die Protokollierung der Scriptblock-Ausführung wird in den Gruppenrichtlinien aktiviert

Beispiel

Der folgende Befehl gibt alle Einträge mit der ID 4104 aus dem Eventlog zurück, die innerhalb der letzten Minute geschrieben wurden.

```
Get-WinEvent -FilterHash @{LogName="Microsoft-Windows-
PowerShell/Operational";Id="4104";StartTime=(Get-Date).AddMinutes(-1)} |
Format-List Message
```

Die Auswertung der Einträge wird durch den Umstand etwas erschwert, dass wie üblich relativ viele Einträge entstehen. Bereits die Ausführung eines Scriptblocks, der lediglich das *Get-Date*-Cmdlet ausführt, hat zwei Einträge zur Folge:

```
Message : ScriptBlock-Text (1 von 1) wird erstellt:
          { Get-Date }
          ScriptBlock-ID: bf6082b3-fbd2-426d-a271-01a2cee7f580
          Pfad:
Message : ScriptBlock-Text (1 von 1) wird erstellt:
          &{ Get-Date }
```

Wird auch das Starten und Beenden eines Scriptblocks geloggt, hat dies jeweils einen Eintrag mit der ID 4105 und 4106 zur Folge. Diese Option sollte nur in Ausnahmefällen aktiviert werden, da sie relativ viele Einträge zur Folge hat, die zudem keine „interessanten" Details liefern.

- **Tipp** Ein EWT-Log kann nicht über das *Clear-EventLog*-Cmdlet geleert werden, sondern entweder über das Befehlszeilentool *wevtutil.exe* oder über die .NET-Klasse *System.Diagnostics.Eventing.Reader.EventLogSession* und deren statische Eigenschaft *GlobalSession*. Das Objekt dieser Eigenschaft besitzt eine Methode *ClearLog()*, der der Name des Eventlogs übergeben wird. Etwas einfacher geht es mit *wevtutil.exe*. Der folgende Befehl leert das Operational-Log der PowerShell:

  ```
  Wevtutil.exe cl Microsoft-Windows-PowerShell/Operational
  ```

- **Tipp** Zu den Beispielskripten für dieses Buch gehört auch ein Skript mit dem Namen *ScriptBlockLoggingViewer.ps1*, das die Auswertung des Scriptblock-Loggings im Rahmen einer Benutzeroberfläche vereinfacht.

15.9.2 Mehr Möglichkeiten bei Start-Transcript

Das *Start-Transcript*-Cmdlet, mit dessen Hilfe die Ein- und Ausgaben einer Sitzung aufgezeichnet werden, wurde mit der Version 5.0 verbessert und erweitert. Die wichtigste Neuerung ist, dass das Cmdlet auch in der ISE funktioniert. Anstelle von Text werden strukturierte Informationen geschrieben, die aus einem Kopf bestehen, der alle wichtigen Details über den Host enthält. Über den *OutputDirectory*-Parameter muss lediglich ein Verzeichnispfad angegeben werden, der Dateiname wird von der PowerShell gebildet. Eine weitere Neuerung besteht darin, dass die Aufzeichnung über eine Gruppenrichtlinie gesteuert werden kann. Auf diese Weise kann sie bei Bedarf ein- und wieder ausgeschaltet werden. Der zugrunde liegende Registry-Schlüssel ist *HKLM:\Software\Policies\Microsoft\Windows\PowerShell\Transcription*, der DWORD-Eintrag *EnableTranscripting* erhält den Wert 1.

Beispiel

Der folgende Befehl startet eine Protokollierung per *Start-Transcript* und legt fest, dass vor jedem Befehl auch die Ausführungszeit protokolliert wird.

```
Start-Transcript –Path C:\Pslog.txt –IncludeInvocationHeader
```

15.10 PowerShell-Remoting-Endpunkte sichern mit Just Enough Administration (JEA)

JEA für „Just Enough Administration" ist eine Technik, die im Zusammenhang mit PowerShell-Remoting zu mehr Sicherheit in Unternehmensnetzwerken führen soll. JEA macht das Einrichten eingeschränkter Endpunkte für Anwender ohne ein Administratorkonto einfach und führt zusätzlich ein rollenbasiertes Berechtigungssystem ein. Basierend auf einer Capability-Datei wird einem Benutzerkonto im Rahmen einer Remoting-Session auf der Grundlage seiner Gruppenzugehörigkeit ein im Rahmen der Datei festgelegter Befehlssatz zugeordnet.

Zwei Dinge sind bei JEA am Anfang wichtig. 1. JEA ist seit Version 5.0 ein fester Bestandteil des WMF und steht damit zum Beispiel auch unter Windows Server 2008 R2 zur Verfügung. 2. JEA bezieht sich ausschließlich auf PowerShell-Remoting. Wer kein PowerShell-Remoting verwendet, benötigt daher kein JEA.

Die Grundüberlegung für die Erfindung von JEA war, dass es in nahezu jedem größeren Unternehmensnetzwerk zu viele Administratorkonten und damit viele unnötige Angriffspunkte gibt. PowerShell-Remoting soll keine Notwendigkeit für ein weiteres Administratorkonto und damit ein weiterer Unsicherheitsfaktor im Unternehmensnetzwerk sein. Jeder, der sich mit einem Administratorkonto anmelden kann, kann natürlich theoretisch per PowerShell-Remoting auf andere Computer im Netzwerk zugreifen und mit seinen Administratorberechtigungen beliebige Befehle auf diesen Computern ausführen. Da das Deaktivieren von PS-Remoting aus der Sicht von Microsoft keine Option sein soll, hat man sich mit JEA eine Technik überlegt, über die sich auf einem Remote-Computer eingeschränkte Endpunkte einrichten lassen, für die kein Administratorkonto erforderlich ist. Trotzdem können Befehle ausgeführt werden, die normalerweise ein Administratorkonto erfordern. Diese Vorgehensweise wird allgemein als *Least Privilege*-Ansatz bezeichnet. Nach diesem Ansatz wird einem Benutzerkonto nur das für die Durchführung einer Aufgabe erforderliche Minimum an Privilegien verliehen. Um dennoch bestimmte vom Administrator zugelassene Aktivitäten durchführen zu können, die eine Administratorberechtigung erfordern, wie zum Beispiel der Neustart eines Dienstes, wird durch JEA ein virtuelles Systemkonto angelegt, über das diese Aktivitäten für die Dauer der Remoting-Session ausgeführt werden können.

Ein weiterer Aspekt bei JEA ist, dass sich eingeschränkte Endpunkte auch per DSC auf andere Computer im Netzwerk verteilen lassen. Bei einer Pull-Konfiguration „zieht" sich

jeder Computer in regelmäßigen Intervallen von seinem Pull Server die Konfigurationsdaten erneut, so dass ein JEA-Endpunkt leicht an die Erfordernisse angepasst werden kann.

Mit JEA gehen bei einer Remoting-Sessionkonfiguration ein paar kleinere Neuerungen einher. Eine davon ist eine Capability-Datei, in der die Möglichkeiten festgelegt werden, die im Rahmen der Remoting-Session zur Verfügung stehen sollen. Diese Datei ist eine Textdatei mit der Erweiterung *.psrc*. Sie wird, vergleichbar mit einer Sessionkonfigurationsdatei, über das *New-PSRoleCapabilityFile*-Cmdlet angelegt. Die Datei spielt die Rolle einer „Whitelist", in der alles das eingetragen wird, was im Rahmen der Remoting-Sitzung an Befehlen und Möglichkeiten zur Verfügung stehen soll. Vor der Einführung von JEA war dafür die Sessionkonfigurationsdatei selber zuständig. Für die Sessionkonfigurationsdatei gibt es mit *RunAsVirtualAccount*, *RoleDefinitions* und *TranscriptDirectory* neue Einträge.

Auch wenn Microsoft JEA als ein neues Sicherheitsmerkmal von Windows Server 2016 vermarktet wurde, ist es ein Bestandteil von WMF ab der Version 5.0 und steht damit auch auf älteren Windows-Versionen zur Verfügung.

▶ **Hinweis** Die offizielle Anlaufstelle für JEA als Open Source-Projekt ist das GitHub-Portal https://github.com/PowerShell/JEA. Die offizielle Dokumentation befindet sich unter https://msdn.microsoft.com/powershell/jea/overview.

15.10.1 JEA in der Praxis

In diesem Abschnitt wird per JEA ein eingeschränkter Remoting-Endpunkt eingerichtet, über den nur wenige Cmdlets und eine Function zur Verfügung stehen. Damit es einfach bleibt, wird der Endpunkt auf dem lokalen Computer angelegt.

Im ersten Schritt wird ein Benutzerkonto benötigt, das kein Administratorkonto ist. Auch wenn dafür natürlich jedes Benutzerkonto in Frage kommt, wird für diese Übung ein Domänenbenutzerkonto mit dem Namen „JEAUser" angelegt. Es kann einer Gruppe „JEAUsers" hinzugefügt werden, so dass auch eine rollenbasierte Zuordnung von Commands in Abhängigkeit einer Gruppenzugehörigkeit möglich ist. Dieser Aspekt spielt in dieser Übung aber keine Rolle.

Beispiel

Die folgende Befehlsfolge legt ein lokales Benutzerkonto „JeaUser" an.

```
$PwSec = "demo+123" | ConvertTo-SecureString -AsPlainText –Force
New-LocalUser -Name JeaUser -Password $PwSec -ErrorAction Ignore
$NonAdminuser = "JeaUser"
```

Auch wenn es nicht erforderlich ist, sollen alle beteiligten Dateien in einem Modul zusammengefasst werden. Das Modul heißt „JeaDemo".

Beispiel

Die folgenden Befehle legen ein Modulverzeichnis inklusive Manifestdatei an.

```
# Modulverzeichnispfad festlegen
$JEAModulePath = "$env:programfiles\windowspowershell\modules\jeademo"
# Modulverzeichnis anlegen
mkdir -Path $JEAModulePath -ErrorAction Ignore
# Modulmanifestdatei anlegen
New-ModuleManifest -Path "$JEAModulePath\Jeademo.psd1" -Author "Pemo" -
ModuleVersion 1.0
```

Im nächsten Schritt wird im Modulverzeichnis ein Unterverzeichnis *Capabilities* angelegt, in der später die Capability-Datei abgelegt wird:

```
# Unterverzeichnis RoleCapabilities im Modul-Verzeichnis anlegen
mkdir -path "$JEAModulePath\RoleCapabilities" -ErrorAction Ignore
```

Im nächsten Schritt wird per *New-PSRoleCapabilityFile*-Cmdlet die Capability-Datei angelegt. Die Parameter werden über eine Hashtable übergeben:

```
$DemoRoleRoleCapabilityParams = @{
  Author = "Pemo"
  CompanyName = "Posh Admin"
  VisibleCmdlets = "Restart-Service"
  FunctionDefinitions = @{Name="Get-UserInfo"; ScriptBlock={$PSSenderInfo
}}
}
# Capability-Datei für die Rolle DemoRolle anlegen
$CapabilityFilePath = "$JEAModulePath\RoleCapabilities\DemoRolle.psrc"
New-PSRoleCapabilityFile -Path $CapabilityFilePath
@DemoRoleRoleCapabilityParams
```

Der Name der Capability-Datei ist wichtig, da über den Dateinamen die Rollenzuordnung an ein Benutzerkonto oder eine Gruppe durchgeführt wird.

Damit kann die Sessionkonfigurationsdatei angelegt werden. Auch ihre Einstellungen werden über eine Hashtable übergeben:

```
# Einstellungen für die Sessionkonfiguration
$DemoSessionConfigParams = @{
  SessionType = "RestrictedRemoteServer"
  RunAsVirtualAccount = $true
  RoleDefinitions = @{$NonAdminuser = @{RoleCapabilities = "DemoRolle"}}
}

# Sessionkonfigurationsdatei anlegen
New-PSSessionConfigurationFile -Path DemoRolle.pssc
@DemoSessionConfigParams
```

Der entscheidende Aspekt ist die Einstellung *RoleDefinitions*, über die die Sessionkonfiguration mit der Capability-Datei „DemoRolle“ und der darin enthaltenen Einschränkungen verbunden wird.

Damit kann die Sessionkonfiguration registriert werden:

```
# Sessionkonfiguration registrieren
Register-PSSessionConfiguration -Path DemoRolle.pssc -Force -Name
DemoRolle
```

Damit liegt die JEA-Sessionkonfiguration vor und kann beim Herstellen einer Verbindung angegeben werden:

```
# Betreten der eingeschränkten Session auf dem lokalen Computer
$Cred = [PSCredential]::new($NonAdminuser, $PwSec)
Enter-PSSession -ComputerName Localhost -ConfigurationName DemoRolle -
Credential $Cred
```

Ein *Get-Command* macht deutlich, dass mit *Restart-Service* nur ein Cmdlet zur Verfügung steht. Selbst ein Get-Date kann in dieser Session nicht ausgeführt werden. Das Besondere ist aber nicht der Umstand, dass nur wenige Commands zur Auswahl stehen. Das wäre auch ohne JEA möglich gewesen. Das Besondere ist, dass der Benutzer in der Session einen Systemdienst neu starten kann, ohne sich mit einem Administratorkonto anmelden zu müssen.

Ein Aufruf des *Set-PSSessionConfiguration*-Cmdlet mit dem *ShowSecurityDescriptorUI*-Parameter macht deutlich, dass nur noch „JEAUser“ einen Vollzugriff auf den Endpunkt besitzt. Mit anderen Worten: Es kann sich nur dieser eine Benutzer mit diesem Endpunkt verbinden. Er erhält dabei einen Vollzugriff, da die Session über die Capability-Datei eingeschränkt wird (Abb. 15.6 und 15.7). Ein *whoami* gibt an, dass die Session unter einem virtuellen Account läuft.

▶ **Hinweis** Whoami? Dieses Programm kann man in der eingeschränkten Session doch gar nicht ausführen. Das ist richtig, allerdings ist genau dafür die Capability-Datei gedacht. Indem unter „FunctionDefinitions“ eine weitere Function über eine Hashtable definiert wird, in deren Scriptblock zum Beispiel *whoami* ausgeführt wird. Auf diese Weise lassen sich auch beliebige Skripte zur Verfügung stellen, indem sie über eine Function aufrufbar gemacht werden. Das Skript selber ist für den Remote-User nicht erreichbar, solange ihm keine Befehle wie *Get-ChildItem* und *Set-Content* zur Verfügung gestellt werden.

```
PS C:\Users\pemo10\Documents> Enter-PSSession -ComputerName Localhost -ConfigurationName DemoRolle -Credential $Cred

[Localhost]: PS> Get-Command

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Clear-Host
Function        Exit-PSSession
Function        Get-Command
Function        Get-FormatData
Function        Get-Help
Function        Get-UserInfo
Function        Measure-Object
Function        Out-Default
Function        Select-Object
Cmdlet          Restart-Service                                    3.0.0.0    Microsoft.PowerShell.Management


[Localhost]: PS>
```

Abb. 15.6 In der JEA-Session stehen nur ausgewählte Cmdlets zur Verfügung

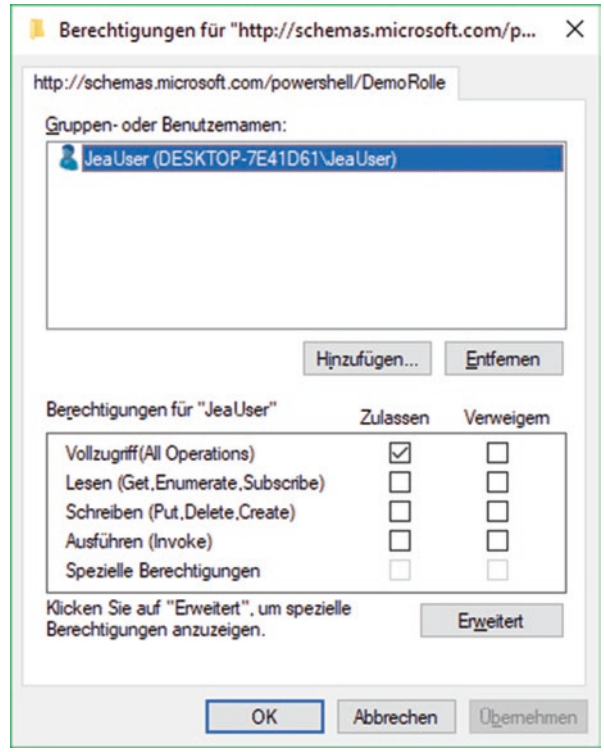

Abb. 15.7 In der Sessionkonfiguration besitzt das Benutzerkonto JEAUser einen Vollzugriff

15.11 Eine Session ohne Remoting einschränken

Die bisher vorgestellten Themen dürften suggeriert haben, dass das Einschränken einer Session bezüglich der zur Auswahl stehenden Befehle nur im Rahmen einer Remoting-Session möglich ist. Das ist nicht der Fall. Auch eine lokale Session kann eingeschränkt werden. Die Einschränkung besteht in diesem Fall darin, dass alle Cmdlets, die nicht zur Ausführung stehen sollen, unsichtbar gemacht und als LanguageMode für die Session „RestrictedLanguage“ gewählt werden. Auch wenn dieser Ansatz nicht perfekt ist, sollte man ihn als erfahrener PowerShell-Anwender kennen.

Beispiel

Das folgende Beispiel ist ein typisches Demo-Beispiel, denn es schränkt die aktuelle Session ein, in dem einfach Cmdlets, die nicht in einem Array enthalten sind, unsichtbar gemacht werden. Die Einschränkung lässt sich daher sehr einfach umgehen. In der Praxis würde man diese Befehle in ein Profilskript einbauen und davon ausgehen, dass der Anwender, der die PowerShell startet, nichts davon weiß oder nicht auf die Idee kommen würde, an dem Profilskript „herumzudoktern“.

```
<#
 .Synopsis
  Einschränkte Session anlegen, indem nicht erwünschte Cmdlets unsichtbar
gemacht werden
#>
$AllowedCommands = @("Get-ChildItem")
$AllowedCommands += "Get-Content"
$AllowedCommands += "Get-Command"
$AllowedCommands += "Get-Process"
$AllowedCommands += "Get-Service"
$AllowedCommands += "Get-FormatData"
$AllowedCommands += "Select-Object"
$AllowedCommands += "Measure-Object"
$AllowedCommands += "Out-String"
$AllowedCommands += "Out-File"
$AllowedCommands += "Out-Default"
$AllowedCommands += "Exit-PSSession"

# Keine Anwendungen, keine Skripte
$ExecutionContext.SessionState.Applications.Clear()
$ExecutionContext.SessionState.Scripts.Clear()

# Alle nicht erwünschten Cmdlets unsichtbar Machen
Get-Command -CommandType Cmdlet | Where-Object { $AllowedCommands -
notcontains $_.Name } |  Foreach-Object {
      $_.Visibility = "Private"
}
$ExecutionContext.SessionState.LanguageMode = "RestrictedLanguage"
Write-Warning "Eingeschränkte Session ist aktiv..."
```

▶ **Hinweis** Bei der Version 2.0 wurde diese Technik im Rahmen von PS-Remoting in ein Startup-Skript eingebaut, das mit dem Herstellen der Verbindung ausgeführt wird. Auf diese Weise stand im Rahmen der Remoting-Session nur ein festgelegter Satz an Cmdlets zur Auswahl. Seit der Version 3.0 gibt es dafür die Sessionkonfigurationsdateien, seit der Version 5.0 gibt es dafür JEA.

15.12 Das Invoke-Expression-Cmdlet und warum es potentiell gefährlich ist

Ein Cmdlet, das in keinem der bisherigen Kapitel explizit vorgestellt wurde, ist das *Invoke-Expression*-Cmdlet. Es ist bereits seit der Version 1.0 dabei. Es interpretiert Text als PowerShell-Befehle und führt diese aus. Das *Invoke-Expression*-Cmdlet ist potenziell gefährlich, da es beliebigen PowerShell-Code ausführt und dies nicht verhindert werden kann. Erst mit den erweiterten Logging-Möglichkeiten, die mit Version 5.0 eingeführt wurden (beziehungsweise mit dem November-Update für die Version 4.0), kann zumindest nachvollzogen werden, dass PowerShell-Befehle auf diese Weise ausgeführt wurden.

Beispiel

Der folgende Befehl lädt das Modul *Open-SSH*, das unter GitHub zur Verfügung steht, über eine Ps1-Datei, die über die *Webclient*-Klasse und ihrer *DownloadString()*-Methode

als Zeichenkette heruntergeladen wird. Diese wird anschließend über das *Invoke-Expression*-Cmdlet ausgeführt.

```
iex (New-Object
Net.WebClient).DownloadString("https://gist.github.com/darkoperator/6152630/raw/c67de4f7cd780ba367cccbc2593f38d18ce6df89/instposhsshdev")
```

Diesen Befehl wird man im Allgemeinen nicht eintippen, sondern von der Webseite kopieren und in die PowerShell-Konsole einfügen.

▶ **Tipp** Da man im Allgemeinen ungerne irgendwelche Skripte ausführen lassen möchte, sollten Sie sich die Ps1-Datei im Browser anschauen, indem Sie den Pfad der Ps1-Datei in das Adressfeld einfügen.

Das Skript, welches das *Posh-SSH*-Modul anlegt, lädt per *WebClient*-Objekt eine Zip-Datei herunter, packt sie mit den Mitteln des Explorers aus und kopiert den Inhalt in ein Modulverzeichnis, das zuvor angelegt wurde. Am Ende wird das Modul importiert und seine Commands ausgegeben. Alles harmlose Befehle, die in einem Rutsch ausgeführt werden.

```
$webclient = New-Object System.Net.WebClient
$url = https://github.com/darkoperator/Posh-SSH/archive/master.zip
Write-Host "Downloading latest version of Posh-SSH from $url" -
ForegroundColor Cyan
$file = "$($env:TEMP)\Posh-SSH.zip"
$Webclient.DownloadFile($url,$file)
Write-Host "File saved to $file" -ForegroundColor Green
$targetondisk = "$($env:USERPROFILE)\Documents\WindowsPowerShell\Modules"
New-Item -ItemType Directory -Force -Path $targetondisk | out-null
$shell_app=new-object -com shell.application
$zip_file = $shell_app.namespace($file)
Write-Host "Uncompressing the Zip file to $($targetondisk)" -
ForegroundColor Cyan
$destination = $shell_app.namespace($targetondisk)
$destination.Copyhere($zip_file.items(), 0x10)
Write-Host "Renaming folder" -ForegroundColor Cyan
Rename-Item -Path ($targetondisk+"\Posh-SSH-master") -NewName "Posh-SSH"
-Force
Write-Host "Module has been installed" -ForegroundColor Green
Import-Module -Name posh-ssh
Get-Command -Module Posh-SSH
```

15.12.1 Invoke-Expression per Scriptblock-Logging überwachen

Dass das *Invoke-Expression*-Cmdlet irgendwelche Befehle ausführt, kann nicht verhindert werden. Dank Scriptblock-Logging lässt sich aber nachvollziehen, ob und welche Befehle ausgeführt wurden. Das Scriptblock-Logging wird im Allgemeinen über eine Gruppenrichtlinie aktiviert (**Computer-Konfiguration -> Administrative Vorlagen -> Windows-Komponenten -> Windows PowerShell**). Wurde die Richtlinie „Protokollierung

von PowerShell-Scriptblöcken aktivieren“ aktiviert, hat jede Ausführung eines Scriptblocks einen Eintrag im PowerShell-Eventlog „Microsoft-Windows-PowerShell/Operational“. Die Event-Id ist 4104.

Beispiel

Das folgende Beispiel muss in der Konsole ausgeführt werden. Es lädt ein kleines Skript von der Webseite des Autors und führt es per *Invoke-Expression*-Cmdlet aus. Das Skript zeigt den Namen des aktuellen Benutzers an. Anschließend wird das Eventlog abgefragt, um sich die Ausführung des Scriptblocks anzeigen zu lassen.

```
$Url = http://www.activetraining.de/Downloads/PoshPayload.ps1

Invoke-Expression -Command ((new-object
net.webclient).DownloadString($Url))
Get-WinEvent -FilterHash @{LogName="Microsoft-Windows-
PowerShell/Operational";Id="4104";StartTime=(Get-Date).AddMinutes(-1) }}
```

Die Ausgabe des Eventlogs zeigt den vollständigen Inhalt des Skripts an, das per *Invoke-Expression*-Cmdlet ausgeführt wurde.

```
Message : ScriptBlock-Text (1 von 1) wird erstellt:
          Add-Type -AssemblyName System.Windows.Forms
          [System.Windows.Forms.MessageBox]::Show("Alles klar,
$env:username")
          ScriptBlock-ID: d09f0b03-22d6-48aa-a968-3df9f3510206
          Pfad:
```

15.13 Zusammenfassung

Sicherheit ist naturgemäß ein Thema mit vielen Facetten. Im Zusammenhang mit der PowerShell sind zwei Aspekte wichtig: Das Verschlüsseln von Texten, so dass sich zum Beispiel Kennwörter sicher ablegen lassen. Und das gezielte Einschränken eines Remoting-Endpunktes. Beide Themen wurden durch Neuerungen adressiert, die mit der Version 5.0 der PowerShell eingeführt wurden.

PowerShell für Linux

16

Zusammenfassung

In diesem kurzen Kapitel geht es um den Einsatz der PowerShell unter Linux am Beispiel einer weit verbreiteten Linux-Distribution Ubuntu.

Alle Hinweise beziehen sich auf eine frühe Vorabversion der PowerShell 6.0 (Alpha 18). Wenn Sie diese Zeilen lesen, dürfte die Installation unter Linux Ubuntu noch etwas einfacher geworden sein und es dürften auch Pakete für andere Linux-Distributionen zur Verfügung stehen. Die offizielle Fertigstellung ist für Ende 2017 anvisiert. Anders als die PowerShell noch ein reines Windows-Zubehör war, gibt es im Projektportal eine Roadmap mit den anvisierten Meilensteinen.

16.1 Ein erster Überblick

Was viele Jahre beinahe utopisch erschien, ist seit dem August 2016 Realität. Microsoft hat die PowerShell auf der Basis von .NET Core neu implementiert und den Quellcode als Open Source-Projekt unter einer MIT-Lizenz freigegeben. Er steht unter dem GitHub-Portal zur Verfügung. Damit niemand den Quellcode kompilieren muss, bietet die GitHub-Projektseite binäre Downloads für die wichtigsten Plattformen in der jeweils aktuellen Version. Stand Mai 2017 lag die kommende PowerShell 6.0 als Alpha-Version vor, eine Beta-Version war für Mai geplant. Für die ersten Schritte sind die Vorabversionen, auch wenn noch nicht alle Funktionalitäten vorhanden sind, optimal geeignet. Mit rudimentären Linux-Kenntnissen lässt sich die PowerShell etwa unter Ubuntu in wenigen Minuten ausprobieren.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_16

Die zentrale Projektseite ist https://github.com/PowerShell/PowerShell. Hier gibt es unter anderem auch eine Liste mit Downloadlinks für die einzelnen Plattformen. Der Quellcode enthält zahlreiche Beispielskripte, in denen auch speziellere Themen wie SSH veranschaulicht werden.

16.2 PowerShell unter Linux installieren

Die PowerShell wird für die verschiedenen Plattformen in Gestalt einer einzelnen Datei zur Verfügung gestellt. Der Download für die Linux-Distribution Ubuntu besteht aus einer Datei mit der Erweiterung *.deb*. Inzwischen stellt das PowerShell-Team eine Registrierung für den lokalen Paketmanager zur Verfügung, so dass die PowerShell wie jedes andere Paket auch per apt-get installiert und vor allem aktualisiert werden kann:

```
sudo apt-get update
sudo apt-get install -y powershell
```

Die Anleitung finden Sie auf der Projektwebseite.

▶ **Hinweis** Das Voranstellen des *sudo*-Kommandos ist unter Linux immer dann erforderlich, wenn ein Programm mit *root*-Rechten ausgeführt werden muss.

Nach der Installation wird die PowerShell im Terminal-Fenster durch Eingabe von „.powershell“ gestartet. Ein *$PSVersionTable* gibt die Versionsnummer und andere Details aus. Der wichtigste Unterschied in diesem Zusammenhang ist, dass es sich um die „Core Edition“ handelt (Abb. 16.1).

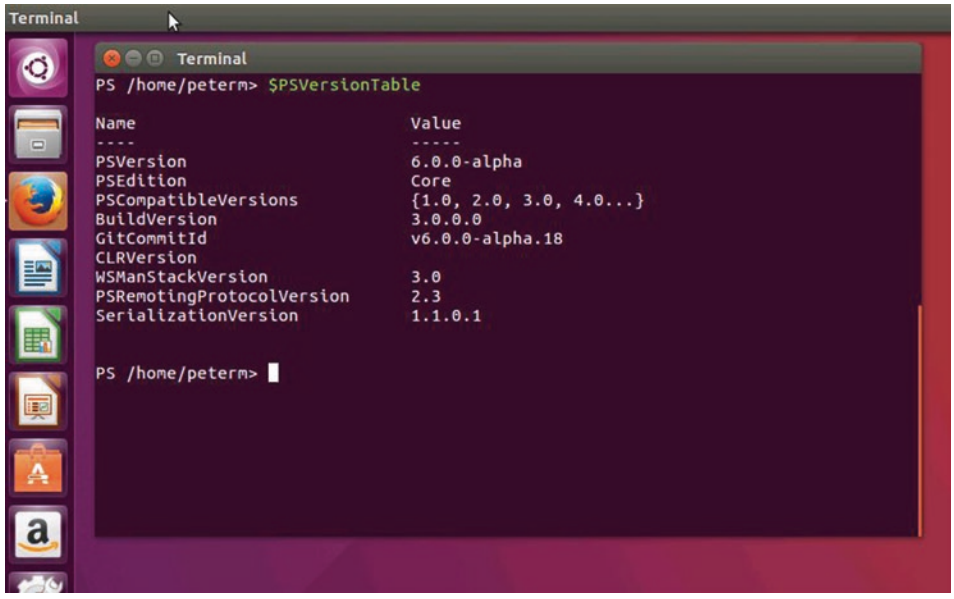

Abb. 16.1 Die PowerShell 6.0 Core unter Ubuntu

16.3 Die ersten Schritte unter Linux

Die ersten Schritte mit der PowerShell unter Linux unterscheiden sich grundsätzlich nicht von den ersten Schritten mit einer PowerShell unter Windows. Auch wenn die Linux-Version weitestgehend kompatibel zur Windows-Version ist, stehen natürlich nicht alle Befehle zur Verfügung. Ein „Get-Service“ kann unter Linux zu keinem Ergebnis führen, da es keine Systemdienste gibt. Genauso wenig gibt es ein „Get-Hotfix“. Möglicherweise lässt sich das PowerShell-Team für die endgültige Version etwas einfallen, so dass keine Fehlermeldungen entstehen.

Die gute Nachricht: Die Groß-/Kleinschreibung spielt auch in dieser Umgebung für PowerShell-Commands keine Rolle. Ein „dir“ oder ein „Dir“ sind daher gleichwertig. Interessant ist, dass jene Alias, die unter Windows ein wenig exotisch gewirkt haben dürften, in der neuen Umgebung perfekt passen. Ein „pwd“ gibt den aktuellen Pfad aus, ein „cat“ den Inhalt einer Datei und ein „ls“ listet den aktuelle Verzeichnisinhalt auf. Die Eingabe von „ps“ liefert eine Liste der laufenden Prozesse. Dies ist allerdings kein PowerShell-Alias, sondern genau wie *ls* ein natives Unix-Kommando. Bei den nativen Kommandos kommt es natürlich auf die Groß-/Kleinschreibung an. Auch die Befehlsschalter werden 1:1 verwendet, zum Beispiel in der Bash-Shell. Wichtig: Das Stammverzeichnis heißt unter Linux / und nicht \.

16.4 Navigieren im Dateisystem

Für Linux-Neulinge ist es sehr hilfreich, dass sich mittels der vertrauten PowerShell-Cmdlets etwas leichter im Dateisystem eines Linux-Computers navigieren lässt als mit den hauseigenen Bash-Kommandos.

Beispiel

Der folgende Befehl gibt alle Verzeichnisse ausgehend vom Stammverzeichnis / aus.

```
dir –Directory –Recurse –Path /
```

16.5 PowerShell-Remoting mit SSH

Unter Windows basiert PowerShell-Remoting bekanntlich auf Ws-Man. Ein Standard, der in der Unix- und Linux-Welt so gut wie unbekannt ist, wenngleich Microsoft bereits vor einigen Jahren eine quelloffene Implementierung eines „Open Management Instrumentation“-Kerns, eine Art „Mini-WMI“, zur Verfügung gestellt hat. Der Standard in der Unix- und Linux-Welt für Remote-Aufrufe ist *SSH* („Secure Shell“). Damit Computer A einen Remote-Aufruf auf Computer B durchführen kann, benötigt Computer A einen SSH-Client und Computer B einen SSH-Server. Beide sind in der Unix-Welt in der Regel von Anfang an Bestandteil des Betriebssystems. Die Authentifizierung erfolgt über Schlüssel.

Es versteht sich von selbst, dass die PowerShell unter Linux SSH anbieten muss, damit es in der Linux-Welt akzeptiert wird. Das bedeutet aber auch, dass SSH auch in der Windows-Welt als Teil der PowerShell in Zukunft eine wichtige Rolle spielt und früher oder später Ws-Man ablösen wird. Um unter Windows eine SSH-Verbindung mit einem Linux-Computer herstellen zu können, wird ein SSH-Client benötigt. Der Standard-SSH-Client ist ein kleines Programm mit dem Namen „Putty". Das PowerShell-Team hat sich einer Implementierung von *OpenSSH* angenommen und stellt den Quellcode über die Projektseite https://github.com/PowerShell/Win32-OpenSSH zur Verfügung. Damit niemand den Quellcode kompilieren muss, gibt es dankenswerterweise ein Package mit dem Namen „Win32-openssh" auf *Chocolatey.org*, das Open SSH für Windows in Gestalt mehrerer Kommandozeilentools hinzufügt.

Auf der Seite des Linux-Computers muss zum einen sichergestellt sein, dass ein SSH-Server an Bord ist. Außerdem müssen in der Konfigurationsdatei „sshd_config" zwei Änderungen durchgeführt werden:

- Es muss ein *subsystem*-Eintrag hinzugefügt werden.
- Die Passwort-Authentifizierung muss aktiviert werden.

Der *subsystem*-Eintrag fügt die PowerShell hinzu. Der vollständige Pfad ist nicht unbedingt erforderlich:

```
subsystem powershell C:\Program Files\Powershell\6.0.0.18\powershell.exe
-sshs -NoLogo –NoProfile
```

Das Übertragen unverschlüsselter Kennwörter muss aktiviert werden:

```
PasswordAuthentication yes
```

Nachdem die Datei gespeichert wurde, muss der *Ssh*-Dienst neu gestartet werden:

```
sudo service ssh restart
```

▶ **Tipp** Für das Editieren der SSH-Konfigurationsdatei ist der Mini-Editor *Nano* sehr gut geeignet, da er anders als zum Beispiel *vi* etwas einfacher in der Bedienung ist.

Ein direktes Pendant zur PowerShell ISE gibt es für Linux natürlich nicht und wird es auch nicht geben, da die ISE nicht portiert werden soll. Der indirekte Nachfolger für alle Plattformen ist Visual Studio Code, dessen PowerShell-Erweiterung natürlich auch unter Linux funktioniert.

Auf der Windows-Seite muss die PowerShell 6.0 in der Alpha-Version für Windows verwendet werden, da ab dieser Version Cmdlets wie *New-PSSession* über ihre Parameter *Hostname* und *Username* auch SSH unterstützen.

Beispiel

Der folgende Befehl stellt eine Verbindung von einem Windows-Computer zu einem Linux-Computer her.

```
$S1 = New-PSSession -Hostname Ubuntu1 -Username pemo
Enter-PSSession -Session $S1
```

▶ **Hinweis** Aktuell (Stand: Mai 2017) gibt es noch einige Einschränkungen beim Remoting via SSH, die im Laufe der Zeit bis zur offiziellen Freigabe wegfallen werden. Eine gute Übersicht gibt es auf der Projektwebseite: https://github.com/PowerShell/PowerShell/tree/master/demos/SSHRemoting.

16.6 PowerShell versus PowerShell Core

Ein wenig Verwirrung herrscht bezüglich des Umstandes, dass es unter Windows in Zukunft zwei „Geschmacksrichtungen" der PowerShell geben wird. Zum einen die vertraute PowerShell, die auf dem vollständigen .NET Framework basiert, als Teil von WMF. Und eine PowerShell, die lediglich auf .NET Core basiert, und die, wie jede Anwendung, als Zip-Datei heruntergeladen, dessen Inhalt in ein Verzeichnis im Programme-Verzeichnis kopiert wird. Da die PowerShell Core auf allen Plattformen (geplant ist auch eine Implementierung für einen Raspberry Pi-Kleincomputer) laufen soll, wird es bestimmte Befehle (u. a. Get-Service) nicht geben. Es existieren damit in Zukunft zwei PowerShell-Varianten, die sich bezüglich ihres Befehlsumfangs unterscheiden. Das PowerShell hatte im Mai 2017 angekündigt, dass Neuerungen zuerst in PowerShell Core implementiert und danach in die Windows PowerShell übernommen werden. Aktuell sind dies aber nur Gedankenspiele, die stark vom Feedback der Anwender abhängen werden.

16.7 Zusammenfassung

Die PowerShell füllt unter Linux eine Lücke. Auch wenn es keinen Mangel an Skriptsprachen gibt und mit der *Bash* seit Jahrzehnten eine Standard-Shell existiert, die im Prinzip alles kann, bringt die PowerShell auch unter Linux ihre bekannten Stärken ins Spiel. Dazu gehört zum Beispiel eine Vereinfachung sowohl bei Ad-hoc-Abfragen als auch in Skripten, die auf dem Umstand basiert, dass alles ein Objekt ist. Von der PowerShell werden daher nicht nur Windows-Administratoren profitieren, die sich unter Linux auf unbekanntes Terrain begeben müssen, sondern eventuell auch erfahrene Linux-Administratoren.

Wie man gute Skripte schreibt

17

Zusammenfassung

Einfache Regeln tragen dazu bei, dass Skripte besser lesbar und besser strukturiert werden und damit besser erweiterbar sind. Und das nicht nur vom Autor des Skripts, sondern auch von anderen PowerShell-Anwendern.

17.1 Kommentare

Jedes Skript sollte mit einem mehrzeiligen Kommentar beginnen. Mit *.Synopsis*, *.Description*, *.Notes* stehen Schlüsselwörter zur Verfügung, über die die folgenden Zeilen automatisch in die Hilfe zu dem Skript eingefügt werden. Es gibt weitere Schlüsselwörter, die in der Hilfe unter „about_comment_based_help“ zusammengefasst sind.

17.2 Variableninitialisierung erzwingen

VBScript-Anwender kennen den Befehl „Option Explicit“. Er bewirkt, dass jede Variable offiziell mit dem Dim-Befehl deklariert werden muss. Ein direktes Pendant dazu gibt es bei der PowerShell nicht. Was dem Befehl nahekommt ist das *Set-StrictMode*-Cmdlet. Es bewirkt, dass Variablen, die keinen Inhalt besitzen, zu einem Fehler führen, wenn die Variablen angesprochen werden. Damit wird verhindert, dass kein Skript Variablen verwendet, die keinen Wert besitzen. Solche Situationen können unter anderem dadurch entstehen, dass ein Variablenname falsch geschrieben wurde.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_17

Beispiel

Der folgende Befehl bewirkt, dass Variablen ohne Inhalt zu einem Fehler führen.

```
Set-StrictMode –Version 2.0
```

Durch den *Version*-Parameter wird die Versionsnummer der PowerShell festgelegt, auf die sich die Regelüberprüfung beziehen soll. Da die Version 1.0 voreingestellt ist und bei dieser Version nicht-initialisierte Variablen innerhalb von Zeichenketten nicht zu einem Fehler führten, wird der Parameter immer mit dem Wert 2.0 angegeben. Alternativ kann auch ein „Latest“ als Wert angegeben werden.

17.3 Aliase vermeiden

Aliase sollten in Skripten nicht verwendet werden, da sie die Lesbarkeit herabsetzen. Wenn ein Skriptautor für lange Befehlsnamen Aliasnamen verwenden möchte, sollten diese zu Beginn des Skripts definiert werden.

17.4 Datentypen für Parameter

Die Parameter von Skripten und Functions sollten immer mit einem Datentyp deklariert werden. Wird kein Datentyp explizit angegeben, erhält der Parameter den allgemeinen Typ *Object*. Die explizite Angabe eines Datentyps wie *String*, *DateTime* oder *PSCredential* bietet mehrere kleinere Vorteile. Der Datentyp trägt zur besseren Lesbarkeit der Deklaration bei. Diese wird auch in die Hilfe übernommen. Wird beim Aufruf ein unpassender Datentyp übergeben, führt der Aufruf zu einem Fehler und das Skript oder die Function werden nicht mit dem unpassenden Typ ausgeführt.

Gibt ein Skript oder eine Function Werte zurück, sollte deren Typ explizit angegeben werden. Dies geschieht über ein dafür vorgesehenes Schlüsselwort der Hilfe mit dem Namen „.Outputs“.

17.5 Keine „Spuren“ hinterlassen

Skripte sollten keine globalen Variablen verändern, ohne sie am Ende wieder auf ihren alten Wert zurückzusetzen. Leider gibt es dafür keinen Automatismus, so dass es darauf hinausläuft, die Werte von Variablen wie *VerbosePreference* einer Variablen zuzuweisen, wenn die Variable durch das Skript einen anderen Wert erhält. Am Ende wird der zwischengespeicherte Wert wieder zugewiesen:

```
$OldVerbosePref = $VerbosePreference
$VerbosePreference = "Continue"


$VerbosePreference = $OldPreference
```

Wer die PowerShell ISE verwendet, kann die Befehlsfolge mit der *New-ISESnippet*-Function dauerhaft speichern und als Ausschnitt per [Strg]+[J] in jedes neue Skript einfügen.

17.6 Der PSScriptAnalyzer

Der *PSScriptAnalyzer* ist ein Modul, mit dessen Hilfe sich Skripte anhand von Regeln überprüfen lassen. Der *PSScriptAnalyzer* stammt von Microsoft und wird als Open Source-Projekt unter Mitwirkung der PowerShell-Community weiterentwickelt. Die aktuelle Version (Stand: Mai 2017) ist 1.11.

Das Modul *PSScriptAnalyzer* umfasst nur zwei Cmdlets: *Get-ScriptAnalyzerRule und Invoke-ScriptAnalyzer*. Über das *Invoke-ScriptAnalyzer* wird ein Skript überprüft. *Get-ScriptAnalyzerRule* gibt alle verfügbaren Regeln als *RuleInfo*-Objekte zurück.

Beispiel

Der folgende Aufruf des *Invoke-ScriptAnalyzer*-Cmdlets überprüft ein Skript und gibt alle Regelverstöße als Objekte vom Typ *DiagnosticRecord* aus.

```
Invoke-ScriptAnalyzer -Path .\Profile.ps1
```

Über den Parameter *Severity* wird der Schweregrad festgelegt, so dass zum Beispiel keine Warnungen ausgegeben werden. Zur Auswahl stehen „Error“, „Warning“ und „Information“.

Beispiel

Der folgende Aufruf des *Invoke-ScriptAnalyzer*-Cmdlets gibt nur Fehler aus.

```
Invoke-ScriptAnalyzer -Path .\Profile.ps1 -Severity Error
```

17.6.1 Eigene Regeln definieren

Der *PSScriptAnalyzer* kann um Regeln erweitert werden. Benutzerdefinierte Regeln werden entweder über eine Assembly zur Verfügung gestellt, die in der Regel mit Visual Studio erstellt wird, oder es kommt eine Psm1-Datei in Frage, in der eine oder mehrere Regeln definiert werden. Die Psm1-Datei muss keinen speziellen Aufbau besitzen. Sie besteht aus einer oder mehreren Functions, die Regelverstöße feststellen (wie auch immer das im Einzelnen geschieht) und diese als Objekte vom Typ *Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord* zurückgeben. Auch wenn dies nicht zwingend erforderlich ist, werden Regelverstöße am einfachsten per *AST* („Abstract Syntax Tree“) festgestellt. Das bedeutet konkret, dass ein Scriptblock per AST in seine Bestandteile zerlegt und diese entsprechend der Regel analysiert werden. Das folgende Beispiel veranschaulicht diese Technik.

Beispiel

Das folgende Beispiel zeigt eine Psm1-Datei, in der eine Regel definiert wird, die überprüft, ob in einem Skript Parameter ohne einen expliziten Datentyp verwendet werden.

```
<#
 .INPUTS
 [System.Management.Automation.Language.ScriptBlockAst]
 .OUTPUTS
 [Microsoft.Windows.Powershell.ScriptAnalyzer.Generic.DiagnosticRecord[]]
#>
function Measure-ObjecWithoutDatatype
{
  [CmdletBinding()]

[OutputType([Microsoft.Windows.Powershell.ScriptAnalyzer.Generic.Diagnost
icRecord[]])]
  param(
      [Parameter(Mandatory = $true)]
      [ValidateNotNullOrEmpty()]

[System.Management.Automation.Language.ScriptBlockAst]$ScriptBlockAst
  )
  process
  {
      [ScriptBlock]$Predicate = {
          param ([System.Management.Automation.Language.Ast]$Ast)
          if($Ast.GetType().Name -eq "ParamBlockAst")
          { return $true}
          else
          { return $false }
      }
      $Results = @()
      try
      {
        # Alle Parameter holen
        [System.Management.Automation.Language.Ast[]]$Paras =
$ScriptBlockAst.FindAll($Predicate, $True)

        $ObjectParas = $Paras.Parameters.Where{$_.StaticType.Name -eq
"Object"}
        if ($ObjectParas.Count -ne 0)
        {
          foreach ($ObjectPara in $ObjectParas)
          {
            $Result  = New-Object `
             -Typename
Microsoft.Windows.PowerShell.ScriptAnalyzer.Generic.DiagnosticRecord `
            -ArgumentList "Jeder Parameter sollte einen expliziten
Datentyp besitzen ($($ObjectPara.Extent.Text))", $ObjectPara.Extent,
"ParameterExplicitDataType", "Warning","",1000,$null
             $Results += $Result
          }
        }
        return $Results
      }
      catch
      {
        $PSCmdlet.ThrowTerminatingError($_)
    }
  }
}
Export-ModuleMember -Function Measure-*
```

Beim Aufruf von *Invoke-ScriptAnalyzer* wird das Modulverzeichnis mit der Psm1-Datei über den Parameter *CustomRulePath* eingebunden.

Beispiel

Der folgende Aufruf verwendet eine benutzerdefinierte Regel für die Analyse einer Skriptdatei.

```
Invoke-ScriptAnalyzer -Path .\Test1.ps1 -CustomRulePath 'C:\Program
Files\WindowsPowerShell\Modules\PSCustRule'
```

17.7 Zusammenfassung

Um gute Skripte zu schreiben, müssen lediglich ein paar einfache Vorgaben eingehalten werden. Mit Hilfe des erweiterbaren PSScriptAnalyzers lässt sich jedes Skript anhand eines Regelsatzes überprüfen. Damit lässt sich mit minimalem Aufwand ein Überblick darüber gewinnen, inwieweit ein Skript bestimmte allgemeine Vorgaben wie zum Beispiel ein Verzicht auf Alias umsetzt.

Spaß mit der PowerShell

18

Zusammenfassung

Wer die PowerShell beruflich einsetzt, beschäftigt sich in der Regel mit Themen, die etwas mit Server-Administration oder anderen ernsthaften Dingen zu tun haben. Mit der PowerShell als kleine Programmiersprache, die bei Windows fest eingebaut ist, und mit der .NET-Laufzeit und ihren vielen Möglichkeiten kann man aber auch Spaß haben. Dazu gehören harmlose Späßchen wie das Abspielen eines Ohrwurms aus den 80er-Jahren, eine sprechende Shell, ein Zitatgenerator oder ein Aprilscherz für Kollegen, die Humor haben (müssen). Beginnen möchte ich mit einem eher trockenen Thema, ohne das aber einige der beschriebenen Ideen nicht umsetzbar wären: den Zufallszahlen.

18.1 Zufallszahlen

Die Grundlage für Spiele oder ein Zitat des Tages ist ein Zufallszahlengenerator, der ein oder mehrere Zahlen nach dem Zufallsprinzip auswählt. Bei der PowerShell ist dafür das Get-Random-Cmdlet zuständig, das erstaunlich flexibel ist. Es liefert nicht nur eine Folge von Zahlen, die keine Dubletten enthält, sondern gibt aus einer Liste von Objekten ein oder mehrere Objekte nach dem Zufallsprinzip zurück.

Beispiel

Wir fangen einfach an. Der folgende Befehl liefert eine ganze Zahl zwischen 1 und 10.

```
Get-Random -Minimum 1 -Maximum 10
```


P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_18

Kommt die 10 vor oder kommt sie nicht vor? Finden wir es heraus, indem wir uns zum Beispiel 100 Zufallszahlen ausgeben lassen:

```
(1..100).ForEach{Get-Random -Minimum 1 -Maximum 10}
```

Die 10 ist nicht dabei, das Maximum wird also ausschließend verwendet. Üblicher ist es, den Zahlenbereich über die Pipeline an den *InputObject*-Parameter zu binden. In diesem Fall kann per *Count*-Parameter die Anzahl an Zufallszahlen festgelegt werden.

Beispiel

Der folgende Befehl liefert 10 Zufallszahlen im Bereich 1 bis 49.

```
1..49 | Get-Random -Count 10 | Sort
```

Durch die sortierte Aufgabe fällt auf, dass keine der Zahlen mehrfach vorkommt. Wird für Count 49 übergeben, erhält man folglich die Zahlen von 1 bis 49. Möchte man eine Zahlenreihe, in der einzelne Zahlen mehrfach vorkommen können, muss Get-Random mehrfach aufgerufen werden.

▶ **Hinweis** Sollten die Zufallszahlen aus irgendeinem Grund nicht zufällig genug sein, gibt es per *SetSeed*-Parameter die Möglichkeit, eine beliebige ganze Zahl als Initialisierungswert vorzugeben. Damit wird beim ersten Aufruf des Cmdlets nicht immer dieselbe Zahl generiert.

Richtig praktisch wird das *Get-Random*-Cmdlet, wenn aus einer Liste beliebiger Werte ein oder mehrere Werte ausgewählt werden sollen.

Beispiel

Der folgende Befehl gibt aus der Liste der zwölf Monatsnamen drei Monatsnamen zurück.

```
(1..12).ForEach{Get-Date -Month $_ -Format MMMM} | Get-Random -Count 3
```

Beispiel

Der folgende Befehl gibt aus einer Datei mit Namen pro Zeile genau einen Namen zurück.

```
(Get-Content .\Spieler.txt) | Get-Random
```

Get-Random lässt sich als einfacher Kennwortgenerator verwenden.

Beispiel

Der folgende Befehl gibt eine zufällige Zeichenfolge aus, in der neben Groß- und Kleinbuchstaben auch ein paar Sonderzeichen dabei sind. Dazu trägt der Umstand bei, dass dem Cmdlet auch mehrere Zahlenbereiche übergeben werden, aus denen dann eine Zahl gezogen wird. Dank dem vielseitigen *Join*-Operator wird aus den Zeichen eine Zeichenkette.

```
((48..91),(97..126) | Get-Random -Count 8 | ForEach { [Char]$_ }) -join
""
```

18.2 Farbige Ausgaben

In den Anfangsjahren präsentierte sich die PowerShell-Konsole in der Regel mit weißer Schrift auf einem dunkelblauen Hintergrund. Fehlermeldungen wurden in roter Schrift ausgegeben und waren daher nur schwer bis gar nicht lesbar. Erst mit *PSReadline* kam Farbe ins Spiel. Bei Unix-Shells wie Bash gehörte Farbe von Anfang an dazu, sowohl als Teil des Prompts als auch bei der Ausgabe. Bei der PowerShell ist das anders. Jede Pipeline-Ausgabe besteht aus Objekten. Der „Objektformatierer“ der PowerShell ordnet jeder Ausgabe anhand des Typs der auszugebenden Objekte eine Tabellen- oder Listenformatierung zu, die festgelegte Eigenschaften umfasst. Diese werden in verschiedenen *Format.ps1xml*-Dateien im *PsHome*-Verzeichnis festgelegt. Farbe kommt dabei nicht vor. Die einzige Möglichkeit, eine Ausgabe farbig zu gestalten, bietet das *Write-Host*-Cmdlet mit seinen Parametern *ForegroundColor* und *BackgroundColor* beziehungsweise die direkte Ausgabe über die *Console*-Klasse. Der Nachteil dieser Methode ist bekannt: Es entsteht eine Ausgabe, die weder umgeleitet noch über die Pipeline weiterverarbeitet werden kann.

Beispiel

Die folgende Function gibt die Eckdaten zu den laufenden Prozessen farbig aus, indem einzelne Spalten in unterschiedlichen Farben ausgegeben werden. Über eine frei definierbare Bedingung werden einzelne Zeilen vollständig in einer festgelegten Farbe ausgegeben. Bei Prozessen, die mehr als 100 MB Arbeitsspeicher belegen, wird die Speicherplatzbelegung in Rot angezeigt. Ein Nachteil der bunten Ausgabe ist natürlich, dass es sich nicht um eine Pipeline-Ausgabe handelt und sie daher nicht weiterverarbeitet werden kann.

```
<#
 .Synopsis
 Farbige Ausgabe mit Write-Host
#>

function Out-Color
{
  param([Parameter(ValueFromPipeline=$true)][Object]$InputObject,
        [Scriptblock]$ColorCondition,
        [String]$Color="Yellow")

  begin
  {
      $Ausgabe = "{0,-10}{1,-40}{2,-20}{3,-12}" -f "ID",
"Name","Startzeit", "Speicher"
      Write-Host $Ausgabe
      $Ausgabe = New-Object -TypeName String -ArgumentList "-",80
      Write-Host $Ausgabe
  }
  process
  {
    foreach($Object in $InputObject)
    {
      if (&$ColorCondition)
      {
          $Ausgabe = "{0,-10}{1,-40}{2,-20}{3,-6} MB" -f $Object.ID,
$Object.Name, $Object.StartTime, ([Math]::Round($Object.WS / 1MB,2))
          Write-Host $Ausgabe -F $Color
      }
      else
      {
        $Ausgabe = "{0,-10}"  -f $Object.Id
        Write-Host $Ausgabe -F Green -NoNewline
        $Ausgabe = "{0,-40}" -f $Object.Name
        Write-Host $Ausgabe -F White -NoNewline
        $Ausgabe = "{0,-20}" -f $Object.StartTime
        Write-Host $Ausgabe -F Cyan -NoNewline
        $Farbe = "White"
        if ($Object.WS -gt 100MB)
        {
            $Farbe = "Red"
        }
        $Ausgabe = "{0,-6:n2} MB" -f ([Math]::Round($Object.WS / 1MB,2))
           Write-Host $Ausgabe -F $Farbe
       }
   }
 }
}

# Länger als 24 Stunden laufende Prozesse hervorheben
Get-Process | Out-Color  -ColorCondition { $_.StartTime -lt (Get-
Date).AddHours(-24) }
```

18.2.1 Farbe in der Konsole dank VT100-Unterstützung

Unter Windows 10 und Windows 2016 im Zusammenspiel mit WMF 5.1 gibt es endlich die Möglichkeit, auch eine Pipeline-Ausgabe farbig zu gestalten. Verantwortlich ist aber nicht eine Erweiterung der PowerShell, sondern des Konsolenfensters. Dieses versteht auf einmal auch VT100-Anweisungen, über die im Rahmen einer Terminalemulation die Ausgabe gesteuert wird. Jede Anweisung besteht aus einer Escape-Sequenz. Eine dieser Anweisungen

legt die Vorder- und Hintergrundfarbe für die Ausgabe der nächsten Zeichen fest. Damit muss man nur noch über eine Formatdefinitionsdatei und das *Update-FormatData*-Cmdlet für die Ausgabe eines bestimmten Objekttyps ein neues Tabellenformat definieren und im Rahmen eines Scriptblocks eine Bedingung prüfen, die anhand eines Vergleichswerts über eine Escape-Sequenz eine bestimmte Vordergrundfarbe einstellt.

Beispiel

Das folgende Beispiel ist umfangreich, aber nicht kompliziert. Es ist umfangreich, da es per XML eine aus zwei Spalten bestehende Tabellenformatierung für *Process*-Objekte definiert. In der zweiten Spalte wird der Wert der *WS*-Eigenschaft ausgegeben. Ist der Wert größer 50 MB, wird die Ausgabe per VT100-Escape-Sequenz farbig ausgegeben. Wer viel Zeit hat, erweitert die Definition mit weiteren Spalten oder definiert ein benanntes Format, das über den *View*-Parameter beim *Format-Table*-Cmdlet ausgewählt wird. Sie finden das Skript, zusammen mit den übrigen Beispielen in diesem Buch, in der „Beispielsammlung“, auf die in der Einleitung zu diesem Buch verwiesen wird.

```
<#
 .Synopsis
Farbige Ausgaben durch VT100-Escape-Sequenzen
#>

$FormatData = @'
<Configuration>
  <ViewDefinitions>
  <View>
    <Name>process</Name>
    <ViewSelectedBy>
        <TypeName>System.Diagnostics.Process</TypeName>
    </ViewSelectedBy>
    <TableControl>
        <TableHeaders>
            <TableColumnHeader>
                <Label>Name</Label>
                <Width>40</Width>
                <Alignment>left</Alignment>
            </TableColumnHeader>
            <TableColumnHeader>
                <Label>WS(MB)</Label>
                <Width>10</Width>
                <Alignment>left</Alignment>
            </TableColumnHeader>
        </TableHeaders>
        <TableRowEntries>
            <TableRowEntry>
                <TableColumnItems>
                    <TableColumnItem>
                        <ScriptBlock>

"$($_.ProcessName.Substring(0,1).ToUpper())$($_.ProcessName.SubString(1))
"
                        </ScriptBlock>
                    </TableColumnItem>
                    <TableColumnItem>
                        <ScriptBlock>
                            if ($_.WS -gt 50MB)
                            {
                                $Esc = [Char]0x1b
                                "${Esc}[91m$([Long]($_.WS /
1MB))${Esc}[0m"
                            }
```

```
                            else
                            {
                                [long]($_.WS / 1MB)
                            }
                        </ScriptBlock>
                    </TableColumnItem>
                </TableColumnItems>
            </TableRowEntry>
        </TableRowEntries>
    </TableControl>
  </View>
 </ViewDefinitions>
</Configuration>
'@

$FormatDataPath = Join-Path -Path $PSScriptRoot -ChildPath
"ProcessUpdate.format.ps1xml"

$FormatData | Set-Content -Path $FormatDataPath -Force

Update-FormatData -PrependPath $FormatDataPath
```

18.3 Ein etwas anderer Prompt

Was in der PowerShell-Konsole am linken Rand der Eingabezeile angezeigt wird, wird als Prompt bezeichnet. Der Inhalt des Prompts wird durch die gleichnamige Function festgelegt. Um den Prompt festzulegen, wird lediglich die Function-Definition ausgeführt, die Function selber muss nicht ausgeführt werden. In der Hilfe sind die Hintergründe unter „about_prompt“ ausführlich beschrieben. Dort wird auch der Standardprompt vorgestellt. Um den Prompt zu ändern, muss daher lediglich die *Prompt*-Function neu definiert werden.

Beispiel

Die folgende *Prompt*-Function zeigt einen Prompt an, wie er bei der Unix-Shell Bash üblich ist.

```
<#
 .Synopsis
 Ein Bash-Prompt
 #>

function Prompt
{
 "$($env:username)@$(Hostname)$(cd)$ "
  "`a"
}
```

- **Tipp** Über *$Host.UI.RawUI.WindowTitle* wird ein Text in der Titelzeile des Konsolenfensters angezeigt.

- **Tipp** Über das Steuerzeichen `a wird ein Ton ausgegeben.

18.3.1 Ein farbiger Prompt

Während farbige Ausgaben etwas Vorarbeit erfordern und immer nur für eine vorgegebene Form einer Ausgabe funktionieren, ist ein farbiger Prompt deutlich einfacher umsetzbar. Um einen farbigen Prompt zu erhalten, müssen lediglich die Bestandteile des Prompts einzeln über das *Write-Host*-Cmdlet ausgegeben werden. Pro Ausgabe wird eine andere Farbe festgelegt. Der Parameter *NoNewLine* sorgt jeweils dafür, dass kein Zeilenumbruch ausgegeben wird.

Was im Einzelnen Teil des Prompts sein soll, ist dem Einfallsreichtum jedes Einzelnen überlassen.

Beispiel

Die *Prompt*-Function in dem folgenden Beispiel zeigt Angaben wie den Namen des Computers, den des Anwenders und die Anzahl der laufenden Prozesse farbig an. Verbunden mit einer Angabe, die darüber Auskunft gibt, ob es sich um eine Administratorshell handelt. Außerdem wird das aktuelle Verzeichnis verkürzt angezeigt und die nicht angezeigten Zeichen werden mit Hilfe eines Regex durch ein ~ abgekürzt. Die Abkürzung besteht darin, dass jeder Verzeichnisname bis auf den letzten mit einem Buchstaben abgekürzt wird. Der Laufwerksname und bei UNC-Pfaden das vorangestellte „\\“ bleiben erhalten. Aus einem „C:\Windows\System32\drivers“ wird zum Beispiel ein „C:\w\s\drivers“.

```
<#
 .Synopsis
 Farbiger Prompt mit Pfad-Verkürzung
#>

function Get-ShortPath
{
  param([String]$Path)
  # Heimverzeichnis durch ~ ersetzen - Replace-Methode statt replace-
Operator, damit Zeichen wie \ nicht escaped werden müssen
  $ShortPath = $Path.Replace($HOME, "~")
  # PSProvider-Name entfernen
  $ShortPath = $ShortPath -replace "^[^:]+::", ""
  # Der Regex bricht bei jedem Verzeichnisnamen bis auf den Letzten mit
dem ersten Zeichen ab, das kein \ ist
  # Ausnahmen sind \\ und \\.
  $ShortPath -replace '\\(\.?)([^\\])[^\\]*(?=\\)','\$1$2'
}

function Prompt
{
  $ColorDelim = [ConsoleColor]::DarkCyan
  $ColorHost = [ConsoleColor]::Green
  $ColorPath = [ConsoleColor]::Cyan
  # Alternative: 0x0A7
  Write-Host "$([Char]0x024) " -N -F $ColorDelim
  Write-Host "$(HostName)@($env:UserName)" -N -F $ColorHost
  Write-Host ' {' -n -f $ColorDelim
  Write-Host (Get-ShortPath (Pwd).Path) -n -f $ColorPath
  Write-Host '}' -n -f $ColorDelim
}
```

▶ **Hinweis** Um den Standard-Prompt zurückzuerhalten, muss die PowerShell neu gestartet werden. Das Löschen der *Prompt*-Function führt dazu, dass ein rudimentärer Prompt angezeigt wird.

18.4 Sounddateien abspielen

Für das Abspielen von Sounddateien im Wave-Format gibt es die Klasse *SystemSounds* im Namespace *System.Media* der .NET-Laufzeit. Über ihre *Play*-Methode wird eine Wave-Datei abgespielt, deren Pfad über den *ArgumentList*-Parameter festgelegt wird.

Beispiel

Die folgende Befehlsfolge spielt alle Wave-Dateien im *Windows\Media*-Verzeichnis der Reihe nach ab.

```
<#
 .Synopsis
 Wave-Dateien abspielen
#>
$WavDateien = Get-ChildItem -Path C:\Windows\Media\*.wav
$Anzahl = 0
foreach($Wav in $WavDateien )
{
  $Anzahl++
  Write-Progress -Activity "Windows-Wav-Dateien" -Status "Spiele
$($Wav.Name)" `
   -PercentComplete (($Anzahl / $WavDateien.Count) * 100)
   $Player = New-Object -TypeName System.Media.SoundPlayer -ArgumentList
$Wav.FullName
   $Player.PlaySync()
}
```

Es ist wichtig, dass die Sounddateien über die Methode *PlaySync()* synchron abgespielt werden. Würden Sie stattdessen die etwas naheliegendere *Play()*-Methode verwenden, würde sich ein interessantes Phänomen ergeben: Der per *Write-Progress* angezeigte Fortschrittsbalken würde „durchrauschen", einige Sounddateien würden zeitverzögert abgespielt werden und kurz danach wäre alles wieder vorbei. Der Grund für dieses Verhalten ist, dass *Play()* asynchron funktioniert und die Ausführung des Befehls fortgesetzt wird, während die erste Sounddatei geladen wird. Kurz danach ist die Schleife fertig, die Ausführung des Skripts endet und damit auch die Lebensdauer des *SoundPlayer*-Objekts.

18.4.1 Systemso unds abspielen

Für das Abspielen der über die Systemsteuerung von Windows festgelegten Systemsounds bietet die .NET-Laufzeit die *SystemSounds*-Klasse. Sie bietet wiederum eine Reihe stati-

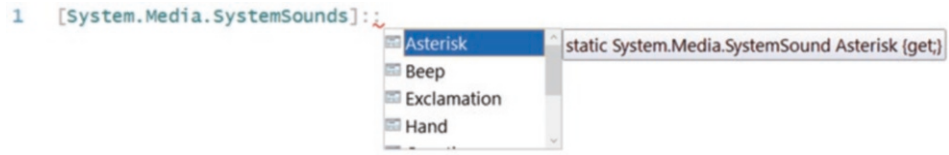

Abb. 18.1 Die PowerShell ISE zeigt die statischen Eigenschaften der SystemSounds-Klasse an

scher Eigenschaften, die jeweils für ein *SystemSound*-Objekt stehen, über dessen *Play()*-Methode die Wave-Datei abgespielt wird.

Der folgende Befehl spielt den Systemsound „Exclamation“ ab (Abb. 18.1):

```
[System.Media.SystemSounds]::Exclamation.Play()
```

18.4.2 Töne erzeugen

Für das Erzeugen von Tönen hat die PowerShell auf den ersten Blick nichts zu bieten. Dabei kann ein akustisches Signal, etwa wenn ein Skript fertig ist oder eine besondere Situation eingetreten ist, auch im administrativen Kontext eine sinnvolle Erweiterung sein, zum Beispiel dann, wenn mehrere Skripte in mehreren Konsolenfenstern parallel auszuführen sind und der Administrator nicht alle Fenster im Auge behalten kann. Für das Erzeugen von Tönen gibt es in der .NET-Laufzeit die statische Methode *Beep()* der *Console*-Klasse, mit der sich einiges anfangen lässt. Ihr werden beim Aufruf immer zwei Argumente übergeben: Das erste Argument bestimmt die Frequenz, das zweite Argument die Abspieldauer in Millisekunden. Da immer nur einzelne Töne ausgegeben werden können, ist damit die typische Computermusik der 80er-Jahre umsetzbar. Akkorde können nicht ausgegeben werden.

Wie die Eingabe des Methodennamens ohne ein rundes Klammerpaar am Ende zeigt, kann die Methode auf zwei Arten aufgerufen werden, ohne ein Argument und mit zwei Argumenten:

```
[System.Console]::Beep

OverloadDefinitions
-------------------
static void Beep()
static void Beep(int frequency, int duration)
```

In der ersten Variante wird ein „Standardton“ ausgegeben, in der zweiten Variante werden Frequenz und Dauer des Tones festgelegt. Das eröffnet zahlreiche Möglichkeiten für das Abspielen bekannter Tonfolgen.

Beispiel

Die folgende Befehlsfolge gibt eine kurze Tonfolge aus, die Fans von SciFi-Kinoklassikern (hoffentlich) mit etwas Fantasie bekannt vorkommen dürfte.

```
<#
 .Synopsis
 Abspielen einer Tonfolge - Teil 1
#>

# Unh. Begeg. d. 3ten A. - soll nach Solresol für ein HELLO stehen g' -
a' - f' - f - c'
$Toene = (800,400), (1000,400), (900,400), (400,800), (600,1600)

foreach($t in $Toene)
{
  [System.Console]::Beep($t[0], $t[1])
}
```

Die Grundlage für eine Tonfolge ist ein zweidimensionales Array, das Tonhöhe und Dauer in Millisekunden für jeden Beep enthält.

Beispiel

Wie wäre es mit der Erkennungsmelodie für einen Kinoerfolg aus den 80er-Jahren?

```
<#
 .Synopsis
 Abspielen einer Tonfolge - Teil 2
#>

# Axel F
$Toene = (659,460), (784,340), (659,230), (660, 110), (880, 230),
         (660,230), (580, 230), (660,460), (988,340), (660,230),
         (660,110), (1047,230), (988,230), (784,230), (660,230),
         (988,230), (1320,230), (660,110), (590,230), (590,230),
         (494,230), (740,230), (660,460)

foreach($t in $Toene)
{
  [System.Console]::Beep($t[0], $t[1])
}
```

Beispiel

Das Beispiel, das im Folgenden vorgestellt wird, verwendet eine einfache Notennotation. Pro Note werden Tonhöhe und Dauer festgelegt. Die einzelnen Noten einer Melodie werden zum Abspielen als Array an eine Function übergeben, die die Noten der Reihe nach per *Beep()*-Methode ausgibt.

```
<#
 .Synopsis
 Abspielen einer Tonfolge - Teil 3
 .Notes - wieder das HELLO in Solresol - dieses Mal mit "echten" Noten
#>

function Play-Melodie
{
  param([Object[]]$Melodie)
  foreach($Tone in $Melodie)
  {
    if ($Tone[0] -eq "Pause")
    {
        Start-Sleep -Milliseconds $Tone[1]
    }
    else
    {
        [System.Console]::Beep($Tone[0], $Tone[1])
    }
  }
}

$Tones =
@{GBelowC=196;A=200;ASharp=233;B=247;C=262;CCharp=277;D=294;DSharp=311;E=
330;F=349;FSharp=370;G=392;GSharp=415}

$Duration = @{WHOLE=1600;HALF=800;QUARTER=400;EIGHTH=200;SIXTEENTH=100}

# Unh. Begeg. d. 3ten A. - soll nach Solresol für ein HELLO stehen g' -
a' - f' - f - c'
$Melodie  = ($Tones.G, $Duration.QUARTER), ("Pause", 400), ($Tones.B,
$Duration.HALF), ($Tones.F, $Duration.QUARTER), ($Tones.F,
$Duration.QUARTER), ($Tones.C, $Duration.QUARTER)

Play-Melodie -Melodie $Melodie
```

18.4.3 PowerShell-Musik

Musikstücke werden bekanntlich durch Noten beschrieben, nicht durch Frequenzangaben. Im TechNet Script Center steht unter der Adresse http://gallery.technet.microsoft.com/scriptcenter/Start-Song-Play-a-song-0b7c8228 ein Skript mit dem Namen „Start-Song.ps1“ zur Verfügung, das eine Function mit dem Namen *Start-Song* enthält, die nicht nur Notenbezeichnungen aus einer Csv-Datei einliest, sondern auch die dazu gehörige Textzeilen. Das mitgelieferte Beispiel passt zwar nur zu einer bestimmten Jahreszeit, ist aber ein schönes Beispiel dafür, was mit „etwas“ Zeit und viel Spaß am Ausprobieren von Techniken, die nichts mit Systemadministration zu tun haben, mit der PowerShell möglich ist.

18.5 Die PowerShell lernt sprechen

Wenn es unbedingt sein muss, spricht die PowerShell alles, was ihr an Textfolgen übergeben wird. Sie verwendet dabei die Sprachausgabe von Windows, die über die COM-Komponente *SAPI.SpVoice* angesprochen wird. Der Aufruf könnte einfacher nicht sein.

Beispiel

Die folgende Befehlsfolge spricht einen Text mit der angenehm klingenden Stimme „Microsoft Hedda Desktop“.

```
<#
 .Synopsis
 Sprachausgabe per SAPI
#>

$SprachText = "Gehen Sie direkt ins Gefängnis, gehen Sie nicht über Los."
$SPVoice = New-Object -ComObject SAPI.SpVoice

# Mit der Standardsprache ausgeben
$SPVoice.Speak($SprachText)
```

Möchte man eine andere Sprache verwenden, muss man sich zunächst einen Überblick über die vorhandenen Sprachen verschaffen.

Beispiel

Der folgende Befehl gibt alle vorhandenen Sprachen aus.

```
$SPVoice.GetVoices() | ForEach-Object { $_.GetDescription() }
```

Anschließend wird die Stimme nach dem Muster „Name=Bezeichnung“ geholt.

Beispiel

Die folgende Befehlsfolge gibt einen kurzen Text mit der englischen Stimme „Microsoft Zira Desktop“ aus.

```
<#
 .Synopsis
 Sprachausgabe per SAPI
#>

$SPVoice = New-Object -ComObject SAPI.SpVoice
$SPVoice.GetVoices() | ForEach-Object { $_.GetDescription() }

# The English Voice holen
$USVoice = $SPVoice.GetVoices() | Where-Object { $_.GetDescription() -
match "English" }
$USVoiceName = $USVoice.GetDescription()
$USVoiceName = $USVoiceName.Substring(0, $USVoiceName.IndexOf("-") -1)
$USVoice  = $SpVoice.GetVoices("Name=" + $USVoiceName).Item(0)
$SPVoice.Voice = $USVoice
$SprachText = "The NSA knows you very good"
$SPVoice.Speak($SprachText)
```

18.5.1 Die .NET-Laufzeit kann auch sprechen

Die Sprachausgabe kann auch über eine Klasse der .NET-Laufzeit angesprochen werden: Zuständig ist die Klasse *SpeechSynthesizer* aus dem Namespace *System.Speech.Synthesis*. Voraussetzung ist, dass zuvor die Assembly *System.Speech* geladen wurde.

Beispiel

Das folgende Beispiel ist ausnahmsweise kein Spaßbeispiel. Es gibt die Anzahl der laufenden Prozesse aus, die mehr als eine bestimmte Anzahl an MB im Arbeitsspeicher belegen, zusammen mit dieser Angabe aus. Die PowerShell macht es sehr einfach, variable Werte in einer Ausgabe zu kombinieren.

```
$LimitMB = 80
$Anzahl = (Get-Process | Where WS -gt ($LimitMB * 1MB)).Count
$Ausgabe = "$Anzahl Prozesse belegen aktuell mehr als $LimitMB MB"
$Synth.Speak($Ausgabe)
```

18.6 ASCII-Art und die 80er-Jahre

Wer Spaß an den großvolumigen Banner-Ausgaben hat, die früher die Ausdrucke des Rechenzentrumsdruckers zierten, wird auch Spaß an dem Skript *Write-Banner.ps1* von *Oscar Virot* haben, das im TechNet Script Center zur Verfügung steht, und in dem sehr viel Fleißarbeit steckt: http://gallery.technet.microsoft.com/scriptcenter/Write-Banner-A-simple-ca6cc719.

Nach dem Download wird die Ps1-Datei wie üblich „entsperrt“ und anschließend dot-sourced ausgeführt. Das Banner wird über die Function *Write-Banner* ausgegeben.

Beispiel

Der folgende Aufruf gibt einen Begriff als Banner-Text aus.

```
Write-Banner -Object "Cyber 176" -ForegroundColor Black -BackgroundColor
White
```

18.6.1 Bewegte ASCII-Art

Dass sich mit simplen ASCII-Zeichen eine Menge anfangen lässt, macht ein PowerShell-Skript deutlich, dass für ein paar Momente die 80er-Jahre wieder zum Leben erweckt. Wer dieser Musik in der Vergangenheit nichts abgewinnen konnte, bitte unbedingt den Lautsprecher abschalten.

Beispiel

Das folgende Skript bringt *Rick Astley* zum Tanzen. Es wird per *Invoke-Expression* direkt von der Adresse http://bit.ly/e0Mw9w heruntergeladen und direkt ausgeführt. Keine Sorge, dahinter steckt eine seriöse Webseite von PowerShell-Team-Mitglied *Lee Holmes*, der unter anderem Autor des sehr empfehlenswerten PowerShell Cookbooks ist. Die reale Adresse ist http://www.leeholmes.com/projects/ps_html5/Invoke-PSHtml5.ps1.

```
iex (New-Object Net.WebClient).DownloadString("http://bit.ly/e0Mw9w")
```

Nach dem Drücken der [Enter]-Taste dauert es einen kurzen Augenblick und danach beginnt *Rick Astley*, der übrigens im Jahr 2017 wieder ein Album herausgebracht hat, zu tanzen.

18.7 Ein Zitat, bitte

Ein „Quote of the day“ ist ein Standardfeature in vielen Konsolen in der Unix-Welt und ein fester Bestandteil der „Internet 1.0-Kultur“, die viele nur noch aus Erzählungen kennen dürften. Über den UDP-Port 17 wird auf Anfrage ein Zitat des Tages geliefert. Zitatserver gibt es natürlich im Internet, aber auch Windows bringt einen solchen Dienst mit, er muss in der Regel über das Feature „Simple-TCPIP“ nachträglich hinzugefügt werden. Anschließend kann zum Beispiel per Telnet- oder SSH-Client der Port 17 angesprochen werden, das Ergebnis ist ein klassisches Zitat auf Englisch. Dieses stammt nicht aus dem Internet, sondern aus der Datei *quotes* im Verzeichnis *C:\Windows\System32\drivers*etc. Wer möchte, kann die Datei beliebig ergänzen.

Beispiel

Das folgende Beispiel fragt einen lokalen „Zitateserver“ ab. Voraussetzung ist, dass der Systemdienst *Simptcp* ausführt.

```
<#
 .Synopsis
 Quote of the Day Service (QOTD) abfragen
 .Notes
 Der Dienst Simptcp muss ausführen - dieser wird über das Feature
"Einfache TCPIP-Dienste" hinzugefügt
#>

$QOTDHost = "localhost"
$UDPClient = New-Object -TypeName System.Net.Sockets.Udpclient

$UDPClient.Connect($QOTDHost, 17)
$UDPClient.Client.ReceiveTimeout = 1000

# Beliebige Textmeldung an den QOTD-Service schicken
$ACSCII = New-Object -TypeName System.Text.ASCIIEncoding
$ByteBuf = $ACSCII.GetBytes("Test1234")
[void]$UDPClient.Send($ByteBuf, $ByteBuf.Length)

# Verbindung mit dem Endpunkt (localhost - Port 17) herstellen
$RemoteEnd = New-Object -TypeName System.Net.IPEndPoint -ArgumentList
([System.Net.IPAddress]::Any), 0
try
{
  # Byte-Daten abrufen
  $BytesReceived = $UDPClient.Receive([ref]$RemoteEnd)
  # Aus Byte-Folge Text machen
  $Quote = $ACSCII.GetString($BytesReceived)
  # Zitat ausgeben
  $Quote
}
catch
{
  Write-Warning "Fehler beim Abrufen von Daten ($_)"
}

# Verbindung wieder schließen
$UDPClient.Close()
```

18.7.1 Einen Internet-Zeitserver abfragen

Ein weiterer klassischer Internet-Dienst sind Zeitserver, die die aktuelle Uhrzeit zur Verfügung stellen. Dahinter steckt lediglich ein Dienst, der auf Port 13 auf Verbindungsanfragen wartet und als Antwort das aktuelle Datum und die aktuelle Uhrzeit zurückgibt.

Beispiel

Das folgende Beispiel ist ein Skript, das die Adressen einer Reihe von Internet-Zeitservern enthält, die der Reihe nach abgefragt werden. Mit der ersten erfolgreichen Kontaktaufnahme bricht die Anfrage ab. Auch bei diesem Beispiel kommt mit *TCPClient* eine zentrale Klasse der .NET-Laufzeit im Namespace *System.Net* zum Einsatz. Der aus der Abfrage resultierende Byte-Stream wird per *StreamReader*-Klasse in Text umgewandelt, der mit Hilfe eines Regex in seine Bestandteile zerlegt wird, um das Datum und die Uhrzeit zu erhalten.

```
<#
 .Synopsis
 Internet-Time Server direkt abfragen
#>

$TimeServer = "128.138.140.44",
              "64.90.182.55",
              "206.246.118.250",
              "207.200.81.113",
              "128.138.188.172",
              "64.113.32.5",
              "64.147.116.229",
              "64.125.78.85",
              "128.138.188.172"

$OldVerbosePref = $VerbosePreference
$VerbosePreference = "Continue"

$Port = 13

foreach($Server in $TimeServer)
{
  Write-Verbose "Checking $Server..."
  try
  {
    $Sockets = New-Object -TypeName System.Net.Sockets.TcpClient -
ArgumentList $Server, $Port
    $SocketStream = $Sockets.GetStream()
    $Reader = New-Object -TypeName System.IO.StreamReader -ArgumentList
$SocketStream
      $Reader.ReadLine()
      $ServerResponse = $Reader.ReadLine()
      Write-Verbose "Server-Reponse: $ServerResponse"
      [void]($ServerResponse -match "\d+\s+(?<Year>\d{2})-(?<Day>\d{2})-
(?<Month>\d{2})\s+(?<Hour>\d{2}):(?<Minute>\d{2}):(?<Second>\d{2})")
      $Year = "20$($Matches.Year)"
      $Month = $Matches.Month
      $Day = $Matches.Day
      $Hour = $Matches.Hour
      $Minute = $Matches.Minute
      $Second = $Matches.Second
```

```
      $InternetTime = Get-Date -Year $Year -Month $Month -Day $Day -Hour
$Hour -Minute $Minute -Second $Second
      "The current Internet time from Server $Server`: $InternetTime"
      break
  }
  catch
  {
      "No connection with $Server possible - lets try the next one."
  }
}

$VerbosePreference = $OldVerbosePref
```

18.8 Ein Matrix-Style-Bildschirmschoner

Die große Zeit der Bildschirmschoner liegt zwar schon wieder ein paar Jahrzehnte zurück (was in erster Linie daran liegt, dass es keine Röhrenbildschirme mehr gibt), trotzdem ist ein nett gemachter Screensaver immer ein Hingucker, ein Ausdruck für „den Geek im Manne“ und macht vor allem dem Erschaffer Spaß. Einen ausgefallenen Bildschirmschoner im Stile von CMatrix für Unix-Konsolen stellt *Oisin Grehan*, ein PowerShell-Experte der ersten Stunde, auf *PoshCode* zur Verfügung: http://poshcode.org/2412. Nach dem Ausführen der Function *Enable-ScreenSaver* dauert es ein paar Sekunden, bis grüne Buchstaben auf schwarzem Hintergrund über das Konsolenfenster fließen. Per [ESC]-Taste verschwindet der Spuk wieder. Eine beindruckende Leistung; auch die verwendeten Befehlstechniken sind lehrreich.

18.9 HAL ist IBM – der unwiderlegbare Beweis

In Stanley Kubricks Science Fiction-Klassiker „Odyssee im Weltraum“ aus dem Jahr 1968 (!) heißt der emotional einfühlsame Supercomputer an Bord des Raumschiffs HAL 9000. Angeblich ist HAL eine Anspielung auf IBM, das zur Zeit der Entstehung des Films ein Codewort für allumfassende Computerintelligenz gewesen sein dürfte (so wie heute Facebook). Ich habe das bislang nicht geglaubt, bis ich von einem anonymen Hacker den folgenden PowerShell-Befehl erhielt:

```
"HAL ist $(([Byte[]][Char[]]"HAL").ForEach{[Char]($_+1)} -join `"")"
```

18.10 April, April

Humor ist bekanntlich, wenn man trotzdem lacht. Eine sehr einfache, aber wirksame Methode, um einen Kollegen, der ein eifriger PowerShell-User ist, in den April zu schicken, besteht darin, vertraute Cmdlets durch gleichnamige Functions zu ersetzen. Da eine Function bei der Ausführung immer Vorrang hat, wird die Function anstelle des Cmdlets ausgeführt. Wenn die Function statt der erwarteten Ausgabe irgendwelchen Nonsens ausgibt, ist damit der Scherz gelungen. Eine ernst gemeinte Warnung vorweg: Auch wenn der

vorgestellte April-Scherz absolut harmlos ist, bitte probieren Sie ihn nur bei Menschen aus, bei denen Sie sicher sind, dass sie den kleinen Spaß locker nehmen (und die sich vor allem keine Strafaktion bei passender Gelegenheit, nämlich wenn man die ganze Sache bereits vergessen hat, einfallen lassen).

Beispiel

Das folgende Skript ersetzt ein paar vertraute Cmdlets durch gleichnamige Functions, die anstelle einer Rückgabe eine bunte Ausgabe untermalt von Beeptönen in der Konsole ausgeben. Damit sich der Originalzustand der PowerShell-Konsole wiederherstellen lässt, gibt es die Function *Clean-Mess*, die alle definierten Functions wieder entfernt. Ansonsten muss die PowerShell einfach nur neu gestartet werden.

```
# Ein April-Scherz-Beispiel

function Out-Psychedelic
{
  param([String]$Term)
  "Ätsch.. heute gibt es leider keine $Term - bitte wenden Sie sich an
Ihren Administrator".ToCharArray().ForEach{
        $f = "Green","Yellow","Blue","Magenta", "Black" | Get-Random
        Write-Host -ForegroundColor $f -BackgroundColor White -Object $_
-NoNewline
        [Console]::Beep(2000,10)
    }
    Start-Sleep -Milliseconds 200
    [Console]::Beep(800,400)
    [Console]::Beep(400,400)
    [Console]::Beep(2000,400)
    Write-Host
}

function Get-Process
{
    Out-Psychedelic -Term Prozesse
}

function Get-Service
{
    Out-Psychedelic -Term Prozesse
}

function Get-Hotfix
{
    Out-Psychedelic -Term Hotfixes
}

function Get-ADUser
{
  Out-Psychedelic -Term Benutzerkonten
}

function Clean-Mess
{
  @("Process", "Service", "Hotfix", "ADUser").ForEach{
     del function:Get-$_
  }
}
```

18.11 Ein Spielhallenklassiker

Zum Abschluss ein Link auf ein PowerShell-Skript, das den Spielhallen-Klassiker *Space Invaders* zu neuem Leben erweckt: http://ps1.soapyfrog.com/2007/01/02/space-invaders. Das Skript basiert komplett auf den Möglichkeiten der Konsole, kommt ohne Erweiterungen aus und funktioniert theoretisch auch mit der Version 1.0. Ein großes Kompliment an die Autoren des Skriptes.

18.12 Zusammenfassung

Die PowerShell eignet sich auch für „Anforderungen“, die nur indirekt etwas mit dem Thema Administration zu tun haben. Wer Spaß am Ausprobieren hat und sich einfache Programmierkenntnisse aneignet, kann auch mit der PowerShell viel Freude haben. Der Fantasie sind wie so oft kaum Grenzen gesetzt.

PowerShell für Entwickler

19

Zusammenfassung

Die PowerShell kommt nicht nur für Administratoren, sondern auch für (Software-) Entwickler als Allround-Werkzeug in Frage. In diesem Kapitel soll die PowerShell aus der Perspektive eines Software-Entwicklers vorgestellt werden, der mit dem .NET Framework und einer Programmiersprache wie C# oder Visual Basic vertraut ist. Es geht um Gemeinsamkeiten und um kleine Unterschiede zur Programmierung mit C# und anderen .NET-Programmiersprachen. Im Mittelpunkt steht das universelle Add-Type-Cmdlet, das aus der Sicht eines Entwicklers kleine Wunder vollbringt, indem es nicht nur Assembly-Dlls lädt und die enthaltenen Typen zur Verfügung stellt, sondern auch das Kompilieren von C#-Code in eine Dll- oder Exe-Assembly ermöglicht. Die weiteren Themen sind der Umgang mit generischen Typen, der Umgang mit Events, das flexible Typensystem der PowerShell, das Erstellen von Benutzeroberflächen mit WPF und der Aufruf von Win32-Funktionen.

19.1 Unterschiede und Gemeinsamkeiten mit C#

Ein Vergleich zwischen der PowerShell als Skriptsprache und C# ist wie der sprichwörtliche Vergleich zwischen Äpfel und Birnen. Die PowerShell ist eine klassische Interpretersprache mit einem flexiblen Typensystem und wenigen Sprachelementen.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2_19

C# ist eine klassische Compilersprache mit einem statischen Typensystem und vielen Sprachelementen. Es gibt trotzdem einige Gemeinsamkeit und Unterschiede:

- Die PowerShell basiert auf der .NET-Laufzeit (in der Regel 4.x), alle Datentypen und Klassen entsprechen denen von C#.
- Die PowerShell erweitert einige .NET-Typen, so dass sie sich etwas anders verhalten als in einem C#-Programm. Ein Beispiel ist die Klasse *DataRow* aus dem Namespace *System.Data*, an die automatisch die Felder einer Tabelle als Eigenschaften angehängt werden.
- Auch bei der PowerShell besitzt jede Variable einen Typ, auch wenn dieser nicht angegeben werden muss.
- Die PowerShell-Funktionalität ist in einer Assembly mit dem Namen *System.Management.Automation.dll* enthalten.
- Assemblys werden mit dem *Add-Type*-Cmdlet in die PowerShell-Sitzung geladen.
- Ein PowerShell-Befehl wird mit der Eingabe in einen *Abstract Syntax Tree* (AST) umgesetzt und damit auf eine gewisse Weise kompiliert. Bei der Ausführung werden die Elemente des Syntaxbaums Schritt für Schritt abgearbeitet. Der AST kann über die gleichnamige Eigenschaft eines Scriptblocks sichtbar gemacht werden.
- PowerShell-Befehle müssen nicht mit einem Semikolon abgeschlossen werden, es darf aber verwendet werden.
- Auch speziellere .NET-Elemente wie Delegaten, Lambdas und Generics können in einem PowerShell-Skript verwendet werden, wenngleich es dafür nur selten eine Notwendigkeit gibt. Ein Lambda ist ein Standardelement bei der PowerShell, denn jeder Scriptblock ist eine anonyme Methode.
- Threads können in der PowerShell nicht angelegt werden (probieren Sie es erst gar nicht).
- LINQ steht nur über die Basisklassen Enumerable und Queryable im Namespace System.Linq zur Verfügung, da es keine Erweiterungsmethoden gibt. Eine Alternative sind das Where-Object-Cmdlet und die Where-Methode be i Arrays.
- Ein PowerShell-Skript kann nicht in C# oder andere Sprachen übersetzt werden. Es ist lediglich möglich, einen PowerShell-Host in ein .NET-Programm einzubauen, der ein Skript ausführt, das in die Exe-Datei als Ressource eingebettet wurde.

19.2 Umgang mit Assemblys

Der Kern der PowerShell basiert auf einer Reihe von Assemblys. Die wichtigste ist *System.Management.Automation.dll*. Auch wenn der Pfad einer Assembly-Datei für deren Nutzung in der Regel keine Rolle spielt, lässt er sich natürlich abfragen. Doch wie komme ich an diese Assembly heran? Ganz einfach über die Eigenschaft *Assembly*, die jedes (!) Typ-Objekt (vom Typ *RuntimeType*) besitzt. Es steht für die Assembly, in der der Typ definiert ist. Wir brauchen daher nur einen Typ, der in der PowerShell-Assembly definiert ist. Zum Beispiel den Typ *PsObject*.

▶ **Hinweis** Die *RuntimeType*-Klasse ist nicht öffentlich, so dass der Typ nicht direkt angesprochen werden kann. Ein „[RuntimeType]“ führt daher zu einem Fehler.

Beispiel

Der folgende Befehl gibt den vollständigen Pfad der Datei *System.Management.Automation.dll* aus.

```
[PSObject].Assembly.Location
```

Für ein besseres Verständnis des .NET Frameworks lohnt es sich, den Verzeichnispfad etwas genauer zu betrachten: *C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll*. Der Verzeichnispfad *C:\Windows\Microsoft.Net\assembly\GAC_MSIL* ist der Pfad des *Global Assembly Cache*, kurz GAC. Der GAC ist ein von der .NET-Laufzeit angelegtes Verzeichnis, in dem alle Assembly-Dateien abgelegt sind, die über ihren vollständigen Namen geladen werden. Der vollständige Name einer Assembly umfasst den Namen der Assembly-Datei ohne die Erweiterung *.dll*, die Versionsnummer, zum Beispiel 1.0.0.0, eine optionale Kulturinfo oder „neutral“ und den Public Key Token, eine aus 16 Ziffern und Buchstaben bestehende „Zahl“. Es fällt auf, dass der Public Key Token nicht Teil des Dateinamens ist, sondern über den Verzeichnispfad gebildet wird. Der Verzeichnisname „v4.0_3.0.0.0__31bf3856ad364e35“ setzt sich aus der Versionsnummer, der nicht vorhandenen Kulturinfo und dem Public Key Token zusammen.

Typenangaben werden bei der PowerShell immer in eckige Klammern gesetzt. Ein solches Element steht für ein System.*RuntimeType*-Objekt. Dieses Objekt wird auch von der *GetType()*-Methode geliefert, die es bei jedem (!) Objekt gibt.

Beispiel

Der folgende Befehl liefert den Pfad der Assembly-Datei, in der die Klasse definiert ist, auf der das Objekt basiert, das ein *Get-ADUser*-Cmdlet liefert.

```
(Get-AdUser -Identity Administrator).GetType().Assembly.Location
D:\Windows\Microsoft.Net\assembly\GAC_64\Microsoft.ActiveDirectory.Manage
ment\v4.0_6.3.0.0__31bf3856ad364e35\Microsoft.ActiveDirectory.Management.
dll
```

19.3 Assemblys in eine PowerShell-Sitzung laden

Eine Assembly wird über das *Add-Type*-Cmdlet geladen. Entweder wird der absolute Pfad der Dll-Datei über den *Path*-Parameter angegeben oder der vollständige Name der Assembly über den *AssembyName*-Parameter. Anschließend stehen alle in der Assembly definierten Typen in der aktuellen PowerShell-Sitzung zur Verfügung.

19.3.1 Assemblys über ihren Pfad laden

Die einfachste Variante besteht darin, eine Assembly über den absoluten Pfad der Dll-Datei zu laden.

Beispiel

Der folgende Befehl lädt die Assembly *PsInfoLib.dll* über ihren Pfad über das *Add-Type*-Cmdlet und den *Path*-Parameter. Die Assembly muss keinerlei Voraussetzungen erfüllen außer, dass es mindestens eine öffentliche Klasse mit einem öffentlichen Member geben muss.

```
$AssPfad = "C:\Users\pemo10\PsInfoLib.dll"
Add-Type -Path $AssPfad -PassThru
```

▶ **Tipp** Wird beim *Add-Type*-Cmdlet der *PassThru*-Parameter gesetzt, werden die Typen der Assemblys in die Pipeline gelegt.

Einen Verweis auf die geladene Assembly erhält man nur indirekt über das Auflisten aller geladenen Assemblys und einer entsprechenden Abfrage.

Beispiel

Der folgende Befehl weist die bereits geladene Assembly *PsInfoLib.dll* einer Variablen zu.

```
$Ass =
[System.AppDomain]::CurrentDomain.GetAssemblies().Where{$_.Location -
match "PsInfoLib"}
```

Ein Entladen einer Assembly ist beim .NET Framework generell nicht vorgesehen (dazu müsste eine Assembly in eine separate AppDomain geladen werden). Möchte man eine per *Add-Type* geladene Assembly wieder loswerden, muss die PowerShell-Sitzung neu gestartet werden.

19.3.2 Assemblys über ihren Namen laden

Der Name einer Assembly darf nicht mit ihrem Dateinamen verwechselt werden. Der (vollständige) Name einer Assembly besteht aus vier Bestandteilen:

1. Dem Dateinamen ohne Erweiterung
2. Einer Kulturinfo oder neutral
3. Einer Versionsnummer (zum Beispiel 1.0.0.0)
4. Dem Public Key Token

Einen Public Key Token besitzt eine Assembly nur, wenn sie mit einem Schlüssel signiert wurde.

Beispiel

Der folgende Befehl lädt die SMO-Assembly („Shared Management Objects“), die eine Vielzahl von Klassen für die Administration eines Microsoft-SQL-Servers enthält, über ihren vollständigen Namen.

```
Add-Type -AssemblyName "Microsoft.SqlServer.Smo, Version=13.0.0.0,
Culture=neutral, PublicKeyToken=89845dcd8080cc91"
```

Spielt die Versionsnummer keine Rolle, geht es natürlich auch einfacher. In diesem Fall genügt der einfache Name der Assembly.

Beispiel

Der folgende Befehl lädt drei für WPF (*Windows Presentation Foundation*) wichtige Assemblys über ihren Namen in einem Rutsch.

```
Add-Type -AssemblyName PresentationFramework, PresentationCore,
WindowsBase
```

In vielen Blogs finden Sie für das Laden einer Assembly eine andere Herangehensweise. Da es das *Add-Type*-Cmdlet erst seit der Version 2.0 gibt, wurde früher die *LoadWithPartialName*-Methode der *Assembly*-Klasse verwendet. Der Vorteil dieser Methode ist, dass eine Assembly auch über ihren kurzen Namen gefunden wird. Diese Variante sollte nicht mehr verwendet, da sie veraltet ist.

Beispiel

Der folgende Befehl lädt die SMO-Assembly über die *LoadWithPartialName*-Methode.

```
[Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo")
```

19.3.3 Alle geladenen Assemblys auflisten

Ein PowerShell-Host muss keine Managed Code-Anwendung sein. *Powershell.exe* ist eine klassische Windows-Konsolenanwendung nach deren Start ein PowerShell-Host angelegt wird. Die PowerShell ISE ist dagegen eine Managed Code-Anwendung. Über die Klasse *AppDomain* und ihre statische Eigenschaft *CurrentDomain* wird ein *AppDomain*-Objekt zur Verfügung gestellt, das eine *GetAssemblies*-Methode enthält. Diese gibt alle aktuell geladenen Assemblys zurück.

Beispiel

Der folgende Befehl gibt die Eckdaten zu allen geladenen Assemblys aus.

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

Beispiel

Der folgende Befehl gibt nur die Namen und die Versionsnummer aller geladenen Assemblys aus.

```
[AppDomain]::CurrentDomain.GetAssemblies().GetName() | Sort
```

Während im Konsolenhost nach dem Start nur 24 Assemblys geladen sind, sind es in der ISE über 50. Der große Unterschied kommt daher, dass die ISE für ihre Ausführung eine Vielzahl von Assemblys benötigt.

19.3.4 Klassendefinitionen sichtbar machen

Eine Assembly enthält in erster Linie Klassendefinitionen. Über das *GetTypes()*-Member eines Assembly-Objekts werden alle Typendefinitionen einer Assembly geholt. Die Eigenschaft *IsClass* gibt ab, ob es sich um eine Klassendefinition handelt.

Beispiel

Der folgende Befehl gibt die Klassendefinitionen aus, die in der PowerShell-Assembly enthalten sind.

```
[PSObject].Assembly.GetTypes().Where{$_.IsClass}.Name | Sort
```

Der Befehl verwendet anstelle des *Where-Object*-Cmdlets die Erweiterungsmethode *Where{}* für Arrays.

Für die praktische Anwendung sind nur die öffentlichen Klassen der Assembly interessant, da sich nur mit ihnen Objekte bilden lassen.

Beispiel

Der folgende Befehl gibt die Namen aller öffentlichen Klassendefinitionen aus, die in der PowerShell-Assembly enthalten sind.

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

Statt 2927 sind dies immerhin noch 682 Klassendefinitionen.

19.3.5 Den Inhalt einer Assembly sichtbar machen

Möchte man den Inhalt einer Assembly auf einen Blick in einem separaten Fenster angezeigt bekommen, in dem man sich jede Klassendefinition anschauen kann, benötigt man ein separates Programm. Ein beliebtes Tool ist *ILSpy* (Abb. 19.1; Download unter http://ilspy.net). Nach dem Start werden bereits eine Reihe von Assemblys geladen. Um zum Beispiel *System.Management.Automation.dll* betrachten zu können, muss die Datei geladen werden. Da der Pfad im GAC etwas kompliziert ist, gibt es eine Alternative in Gestalt

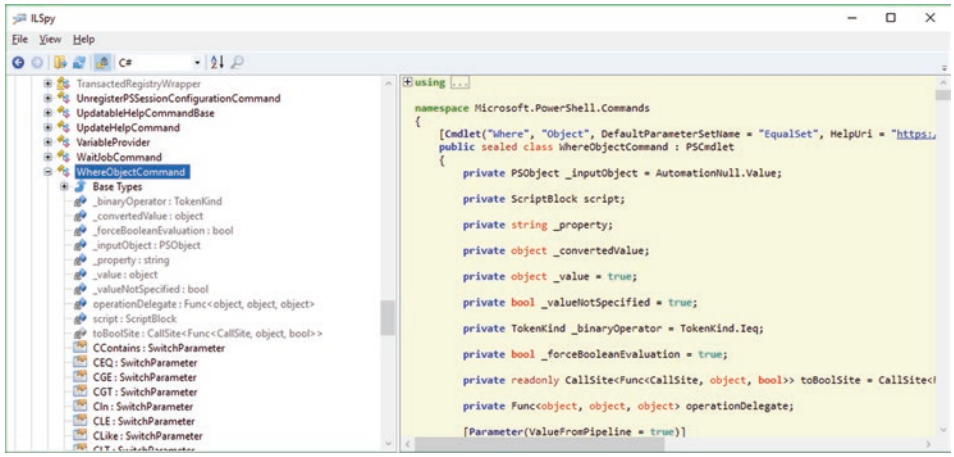

Abb. 19.1 ILSpy zeigt den Inhalt der Assembly System.Management.Automation an

des Verzeichnisses *C:\Program Files (x86)\Reference Assemblies\Microsoft\Windows-PowerShell\3.0* unter einem 64-Bit-Windows.

Beispiel

Die folgende Function erwartet als Argument den Namen eines Typs, zum Beispiel *PSObject*. Sie startet anschließend *Ilspy.exe* und lädt jene Assembly, in der der übergebene Typ definiert ist. Wird *Ilspy.exe* nicht in einem der beiden Programmeverzeichnisse gefunden, wird eine Meldung ausgegeben. Der Name des Typs muss als Argument für den *Type*-Parameter in runde Klammern gesetzt werden. Für den Parameter *Path* kann der Pfad einer Dll-Datei übergeben werden.

```
# Zeigt eine Assembly im ILSpy an
function Show-ILSpy
{
  [CmdletBinding(DefaultParameterSetname="")]
    param([Parameter(Mandatory=$true, Parameterset-
Name="Path")][String]$Path,
          [Parameter(Mandatory=$true, Parameterset-
Name="Type")][Type]$Type)
   # Feststellen, ob ILSpy installiert ist
    if (Test-Path -Path "$env:ProgramFiles\ILSpy\ILSpy.exe")
    {
      $IlspyPath = "$env:ProgramFiles\ILSpy\ILSpy.exe"
    }
    if (Test-Path -Path "$env:ProgramFiles(x86)\ILSpy\ILSpy.exe")
    {
      $IlspyPath = "$env:ProgramFiles(x86)\ILSpy\ILSpy.exe"
    }
    if ($IlspyPath -eq $null)
    {
       throw "Ilspy.exe nicht gefunden - unter http://ilspy.net
installieren"
    }
    if ($PSBoundParameters["Type"])
    {
       $Path = $Type.Assembly.Location
    }
    Start-Process -FilePath $IlspyPath -ArgumentList $Path
}
```

Dank der Function *Show-ILSpy* kann eine Klassendefinition einfach sichtbar gemacht werden. Entweder durch Übergabe des Pfades der Dll-Datei oder indem der Typ als *Type*-Objekt in eckigen Klammern übergeben wird.

Beispiel

Der folgende Aufruf lädt die PowerShell-Assembly.

```
$AssPfad = [PsObject].Assembly.Location
Show-ILSpy -Path $AssPfad
```

Beispiel

Der folgende Aufruf zeigt die Definition der Klasse Ast an.

```
Show-ILSpy -Type ([System.Management.Automation.Language.Ast])
```

Damit die Typenbezeichnung nicht als String interpretiert wird, muss sie noch einmal in runde Klammern gesetzt werden.

19.4 Assemblys erstellen

Warum sollte man in der PowerShell eine Assembly-Bibliothek (oder eine Exe-Datei) erstellen wollen? Ganz einfach, weil es geht und weil es zudem sehr einfach ist. Einen zwingenden Grund dafür gibt es nicht, zumal das Entwicklungswerkzeug Visual Studio unter dem Namen „Community Edition“ seit vielen Versionen kostenlos ist und es mit Visual Studio Code einen Editor gibt, der auf der Grundlage von .NET Core natürlich auch C#-Programme kompilieren kann.

Beispiel

Die folgende Befehlsfolge ist ein PowerShell-Skript in dem C#-Code als Here-String enthalten ist. Per *Add-Type*-Cmdlet und den Parametern *OutputType* und *OutputAssembly* wird der C#-Code in eine Assembly-Dll kompiliert. Das Ergebnis ist eine Datei mit dem Namen *PsInfoLib.dll*. C# ist nur eine Option. Wer Visual Basic bevorzugt, kann auch VB-Code im Skript hinterlegen.

```
<#
 .Synopsis
 Assembly-Bibliothek anlegen
#>

$CSCode = @'
using System;
using System.Management;

public class PsInfo
{
  public string OSVersion { get; set; }
  public string OSCaption { get; set; }
  public long RAM { get; set; }
}
```

```
public class PsSystem
{
  public static PsInfo GetPsInfo()
  {
    // Simuliert eine echte Abfragen per WMI
    string osVersion = "18123.1240";
    string osCaption = "Windows 11 SP2";
    long pcMemory = 549755813888;
    return new PsInfo { OSVersion = osVersion, OSCaption = osCaption,
RAM=pcMemory};
  }
}

'@

Add-Type -TypeDefinition $CSCode -OutputType Library -OutputAssembly
PsInfoLib.dll
```

19.4.1 Herunterladen von NuGet-Packages

Erweiterungen für Entwickler werden in erster Linie als NuGet-Packages zur Verfügung gestellt. Um ein solches Paket per *Install-Package* herunterladen zu können, muss entweder eine Paketquelle eingerichtet oder die URL über den *Source*-Parameter von *Install-Package* direkt angegeben werden:

```
Install-Package –Name JsonNet -ProviderName Nuget -Source
http://www.nuget.org/api/v2 -Force
```

Wird nichts anderes per *Destination*-Parameter festgelegt, wird das Paket unter *C:\Program Files\PackageManagement\NuGet\Packages* abgelegt.

Eine Alternative ist der direkte Aufruf von *Nuget.exe*.

Beispiel

Der folgende Befehl lädt das Package *yamldotnet* herunter und legt es im aktuellen Verzeichnis ab.

```
nuget install yamldotnet
```

19.5 Umgang mit generischen Typen

Die PowerShell kann auch mit generischen Typen souverän umgehen. Ein generischer Typ ist ein Typ, der durch einen Platzhalter (zum Beispiel „T“) repräsentiert wird, zu dem während der Ausführung ein konkreter Typ eingesetzt wird. Im Zusammenhang mit der PowerShell gibt es zwei Berührungspunkte: Generische Listen und Methoden mit generischen Parametern.

19.5.1 Generische Listen

Eine generische Liste ist eine Collection-Klasse, die nur Werte eines beim Anlegen der Collection angegeben Typs aufnehmen kann. Bei der PowerShell wird der Typ für eine generische Liste mit einem „schiefen“ Apostroph angegeben. Ein Beispiel ist die Klasse *System.Collections.Generic.List*.

Beispiel

Das folgende Beispiel legt eine generische Collection vom Typ *List* im Namespace *System.Collections.Generic* an. Es dürfen nur Objekte des festgelegten Typs hinzugefügt werden, in diesem Fall *UserData*, ansonsten resultiert ein Fehler.

```
<#
 .Synopsis
 Generische Listen
#>

class UserData
{
  [Int]$Id
  [String]$Name
}

$UserListe = New-Object -TypeName
System.Collections.Generic.List[`UserData]

# Geht
$UserListe.Add((New-Object -TypeName UserData -Property
@{Id=1000;Name="PemoAdmin"}))

# Geht leider nicht
$UserListe.Add("1234")
```

19.5.2 Generische Methodenaufrufe

Eine Methode einer Klasse kann generische Parameter erwarten, deren Typ zum Beispiel mit dem Instanziieren der Klasse angegeben wird.

Beispiel

Das folgende Beispiel definiert per C#-Code eine Klasse mit einer generischen Methode. Der Aufruf dieser Methode per PowerShell könnte einfacher nicht sein, denn es muss lediglich ein Wert übergeben werden, über den der Typ-Parameter *T* indirekt festgelegt wird.

```
<#
 .Synopsis
 Aufruf einer Methode mit generischen Parameter
#>

$CSCode = @'
  using System;
  using System.Collections.Generic;

  public class GTest
  {

    public static void GetData<T>(T Arg)
    {
      Console.WriteLine("Der Wert ist: {0}, der Typ ist: {1}", Arg,
Arg.GetType().FullName);
    }
  }
'@

Add-Type -TypeDefinition $CSCode -Language CSharp

[GTest]::GetData("Aber Hallo")
[GTest]::GetData(1234)
```

19.5.3 Aufruf einer generischen privaten Methode

Ist die generische Methode privat, wird der Aufruf etwas komplizierter, denn die Methode kann nicht direkt über den Typ bzw. ein Objekt angesprochen werden. In diesem Fall muss die Methode per Reflection gebunden werden.

Beispiel

Das folgende Beispiel definiert wieder eine Methode mit einem generischen Parameter. Der Unterschied zum letzten Beispiel ist, dass die Methode dieses Mal privat ist. Ein solcher Fall tritt bei einigen .NET- und PowerShell-Assemblys aus (wenngleich die Methoden dann keine generischen Parameter verwenden).

```
<#
 .Synopsis
 Aufruf einer Methode mit generischen Parameter
#>

$CSCode = @'
using System;
using System.Collections.Generic;

public class GTest
{
   private static void GetData<T>(T Arg)
   {
      Console.WriteLine("Der Wert ist: {0}, der Typ ist: {1}", Arg,
Arg.GetType().FullName);
   }
}

'@
```

```
Add-Type -TypeDefinition $CSCode -Language CSharp

# Der direkte Aufruf geht nicht
# [GTest]::GetData("Aber Hallo")

# Aufruf muss per Reflection durchgeführt werden

$F1 = [System.Reflection.BindingFlags]::NonPublic
$F2 = [System.Reflection.BindingFlags]::Static

[GTest].GetMethod("GetData", $F1 -bor $F2)

# Nette Abkürzung
$M = [GTest].GetMethod("GetData", @("Static","NonPublic"))
$MG = $M.MakeGenericMethod([String])

# Jetzt fehlt nur noch der Aufruf der Methode
$MG.Invoke($null, "1234")

# Der folgende Aufruf geht dann nicht mehr
$MG.Invoke($null, 1234)
```

Das Geheimnis besteht darin, dass vor dem Aufruf der generischen Methode per Reflection-*Invoke()* der Aufruf der *MakeGenericMethod()*-Methode mit dem Typ des generischen Parameters durchgeführt wird.

Erfahrene C#-Entwickler werden sicherlich beeindruckt davon sein, wie einfach sich die verwendeten Konstanten verwenden lassen. Die binäre Oder-Verknüpfung ist die umständliche Version. Sehr viel einfacher geht es, indem die Namen der Konstanten als Array übergeben werden:

```
$M = [GTest].GetMethod("GetData", @("Static","NonPublic"))
```

19.6 Umgang mit Events

Ein Event ist ein indirekter Aufrufmechanismus, bei dem eine Methode über einen indirekten Aufruf im Rahmen eines Ereignishandlers aufgerufen wird. Events kommen immer dann ins Spiel, wenn ein Skript mit einer grafischen Oberfläche ausgestattet werden soll. Der Event-Mechanismus wird im Framework definiert, aus der Sicht eines Entwicklers ist ein Event ein weiteres Member, das bei der PowerShell vom *Get-Member*-Cmdlet aufgelistet wird. Das setzt allerdings ein Objekt voraus, ohne Objekt muss der Typ direkt angesprochen werden.

Beispiel

Der folgende Befehl gibt die Events der Klasse *Timer* im Namespace *System.Timers* aus.

```
[System.Timers.Timer].GetEvents().Name
```

Um das Einrichten eines Eventhandlers so einfach wie möglich zu gestalten, hängt das Typensystem der PowerShell für jedes Event eine Methode an, der ein Scriptblock übergeben wird. Der Name der Methode beginnt mit „add_“. Ein Beispiel ist das *Elapsed*-Event der Klasse *Timer* im Namespace *System.Timers*. Damit diese Methode(n) per *Get-Member* angezeigt werden, muss der *Force*-Parameter gesetzt werden.

```
$Tmr = New-Object -TypeName System.Timers.Timer
$Tmr | Get-Member -MemberType Method -Force
```

Unter anderem wird das Methodenmember *add_elapsed* ausgegeben.

Auch wenn es theoretisch denkbar wäre, der *addElapsed*-Methode einen Scriptblock zu übergeben, um damit zu erreichen, dass der Scriptblock regelmäßig ausgeführt wird, funktioniert diese Variante in diesem Fall nicht. Der Eventhandler muss offiziell über das *Register-ObjectEvent*-Cmdlet registriert werden. Dabei wird eine beliebige Zeichenkette als „Source Identifier“ übergeben, über die der Eventhandler eindeutig referenziert wird.

Beispiel

Die folgende Befehlsfolge benutzt die *Timer*-Klasse aus dem Namespace *System.Timers*, das *Elapsed*-Event und das *Register-ObjectEvent*-Cmdlet, um zu erreichen, dass alle 5 Sekunden eine Meldung in der Konsole ausgegeben wird.

```
<#
 .Synopsis
 Der "richtige" Umgang mit einem Timer-Event
#>

$SBTimer = {
    $FColor = "Red", "Green", "Yellow", "Cyan" | Get-Random
    Write-Host -f $FColor ("*** Die aktuelle Uhrzeit: {0:HH:mm:ss} ***" -
f (Get-Date))
}

$tmr = New-Object -TypeName System.Timers.Timer
$tmr.Interval = 5000
$tmr.Start()

Register-ObjectEvent -InputObject $tmr -EventName Elapsed -
SourceIdentifier TimerTest -Action $SBTimer
```

Damit der Eventhandler während der PowerShell-Sitzung nicht ewig aktiv ist und Meldungen in die Konsole schreibt, muss er per *Unregister-Event*-Cmdlet und der Angabe des Source Identifiers wieder deregistriert werden:

```
Unregister-Event -SourceIdentifier TimerTest
```

Theoretisch kann der Eventhandler auch im Rahmen des Scriptblocks deregistriert werden. Dazu wird die Variable *$Event* verwendet, die nur im Action-Scriptblock eines Eventhandlers zur Verfügung steht. Sie repräsentiert das aufgetretene Event.

Beispiel

Das folgende Beispiel entspricht dem letzten Beispiel, nur dass die Ausgabe lediglich drei Mal ausgegeben wird.

```
<#
 .Synopsis
 Eventhandler mit Event-Variable
#>

$Anzahl = 0

$SBTimer = {
    $Anzahl++
    if ($Anzahl -eq 3)
    {
        Unregister-Event -SourceIdentifier $Event.SourceIdentifier
    }
    $FColor = "Red", "Green", "Yellow", "Cyan" | Get-Random
    Write-Host -f $FColor ("*** Die aktuelle Uhrzeit: {0:HH:mm:ss} ***" -
f (Get-Date))
}

$tmr = New-Object -TypeName System.Timers.Timer
$tmr.Interval = 2000
$tmr.Start()

Register-ObjectEvent -InputObject $tmr -EventName Elapsed -
SourceIdentifier TimerTest -Action $SBTimer | Out-Null
```

19.7 Das erweiterbare Typensystem

Die PowerShell ist eine klassische Skriptsprache. Als solche geht sie mit Typen flexibler um als C#, bei der ein statisches Typensystem im Mittelpunkt steht. Im Mittelpunkt steht das erweiterbare Typensystem der PowerShell, auch „Extensible Type System“ (ETS) genannt. Das bedeutet konkret, dass es bei der PowerShell einen Typen mit einem festen Satz an Members nicht gibt, da sich ein Typ jederzeit um weitere Members erweitern lässt (das Wegnehmen von Members ist nicht möglich und dürfte auch in der Praxis keine Anforderung sein).

Ein Typ wird über das *Update-TypeData*-Cmdlet erweitert. Die Typendefinitionserweiterung wurde früher in einem XML-Format festgelegt, seit der Version 3.0 bietet das Cmdlet dafür eine Reihe von Parametern.

Beispiel

Das folgende Beispiel zeigt wie einfach es ist, per *Update-TypeData*-Cmdlet einen vorhandenen Typ, wie zum Beispiel *System.IO.FileInfo*, der eine Datei repräsentiert, um ein Member zu erweitern. In diesem Beispiel ist es ein Member mit dem „Comment“, über den einer Datei im Rahmen einer PowerShell-Sitzung ein Kommentar zugeordnet werden kann.

```
<#
 .Synopsis
 Typ-Erweiterung per Update-TypeData
#>

$SB = {

  if ($this.Extension -eq ".Ps1")
  {
   (Get-Help -Name $this.FullName).Synopsis
  }
}

Update-TypeData -TypeName System.IO.FileInfo -MemberName Comment -
MemberType ScriptProperty -Value $SB

dir $env:userprofile\Documents\WindowsPowerShell\PoshBuch | Select Name,
Comment
```

Per *Update-TypeData*-Cmdlet wird ein Typ erweitert. Soll lediglich ein einzelnes Objekt erweitert werden, gibt es dafür das *Add-Member*-Cmdlet.

Beispiel

In dem folgenden Beispiel wird per *[PSCustomObject]*-Typenalias ein Objekt mit einer Eigenschaft „P1“ definiert. Anschließend wird per *Add-Member* eine weitere Eigenschaft „P2“ hinzugefügt.

```
<#
 .Synopsis
 Objekt anlegen und Eigenschaften hinzufügen
#>

$obj = [PsCustomObject]@{P1=100}

# So einfach geht es leider nicht
# $obj.P2 = 200
# Das Hinzufügen eines Members geht nur über Add-Member

$obj | Add-Member -MemberType NoteProperty -Value 200 -Name P2
$obj
```

19.8 Cmdlets definieren

Ein Cmdlet ist nichts anderes als eine Assembly mit einer öffentlichen Klasse, die sich von der *PSCmdlet* oder *PSCmdletInfo*-Klasse im Namespace *System.Management.Automation* ableitet. Auch wenn Cmdlet mit Visual Studio erstellt werden, wenn es unbedingt sein muss, ist dies auch in der ISE oder in Visual Studio Code möglich.

Beispiel

Das folgende Beispiel ist etwas umfangreicher. Das Skript erstellt ein Cmdlet mit dem Namen „Get-Quote“ in Gestalt einer Dll-Datei, das eine Reihe von Zitaten ausgibt,

wenn es mit dem All-Parameter aufgerufen wird. Damit das Beispiel möglichst lehrreich ist, wird auch das Prinzip der Parameter und des Erzeugens von Fehler vom Typ Non Terminating und vom Typ Terminating veranschaulicht.

```
$TypeDef = @'
    using System;
    using System.Management.Automation;

    [Cmdlet(VerbsCommon.Get, "Quote")]
    public class HalloCmdlet : Cmdlet
    {
       private string[] _quotes = {"Eile mit Weile",
                           "Take it easy",
                           "Lerne PowerShell",
                           "Relax",
                           "Dont worry, be happy",
                           "Mach mal urlaub"
                           };
rivate bool _All;
     private bool _ThrowEvilError;
     private bool _NiceError;

     [Parameter(ParameterSetName="Evil")]
     public SwitchParameter ThrowEvilError
     {
        get { return _ThrowEvilError; }
        set { _ThrowEvilError = value; }
     }

     [Parameter(ParameterSetName="Default")]
     public SwitchParameter NiceError
     {
        get { return _NiceError; }
        set { _NiceError = value; }
     }

     [Parameter(ParameterSetName="Default")]
     public SwitchParameter All
     {
        get { return _All; }
        set { _All = value; }
     }

     protected override void ProcessRecord()
     {
        if (_ThrowEvilError)
        {
          string errorId = "HalloWelt.Cmdlet.Error";
          ErrorCategory category = ErrorCategory.InvalidOperation;
          Exception ex =  new InvalidOperationException("Hier ging etwas
gehörig schief");
          ErrorRecord err = new ErrorRecord(ex, errorId, category, null);

          this.ThrowTerminatingError(err);
        }
        else if ( _NiceError)
        {
          string errorId = "HalloWelt.Cmdlet.Error";
          ErrorCategory category = ErrorCategory.InvalidOperation;
          Exception ex =  new InvalidOperationException("Nur eine kleine
Störung, sorry");
          ErrorRecord err = new ErrorRecord(ex, errorId, category, null);
          WriteError(err);
        }
```

```
        if (_All)
        {
           // string allQuotes = String.Join(";",_quotes);
           for(int i =  0; i < _quotes.Length; i++)
              WriteObject(_quotes[i]);
        }
        else
        {
           int i = new Random().Next(0, _quotes.Length);
           WriteObject(_quotes[i]);
        }
     }

    }
'@

Add-Type -TypeDefinition $TypeDef -Language CSharp -ReferencedAssemblies
System.Management.Automation -OutputAssembly HalloCmdlet.dll -OutputType
Library
```

Das Ergebnis ist eine Datei mit dem Namen „HalloCmdlet.dll“, in der die Cmdlet-Definition enthalten ist. Wie wird das Cmdlet geladen? Etwas ungewöhnlich über das *Import-Module*-Cmdlet. Das Ergebnis ist ein binäres Modul:

```
Import-Module .\HalloCmdlet.dll
```

Anschließend kann das Cmdlet *Get-Quote* wie jedes andere Cmdlet auch aufgerufen werden.

19.9 Benutzeroberflächen mit WPF

Ein PowerShell-Skript mit einer Benutzeroberfläche auszustatten, ist grundsätzlich nicht schwer. Die größte Herausforderung für einen Admin, der kein Entwickler ist, besteht darin, das Prinzip zu verstehen und einen Einstieg zu finden. Es gibt sehr viel Material im Internet, unzählige Tutorials und viele Beispiele. Die Fülle des Angebots macht die Übersichtlichkeit nicht unbedingt besser, zumal viele Beispiele bereits einige Jahre alt und teilweise ein wenig umständlich sind.

Für die Umsetzung einer Benutzeroberfläche für ein PowerShell-Skript gibt es zwei Techniken, die beide ein fester Bestandteil der .NET-Laufzeit sind:

- Windows Forms (WinForms)
- Windows Presentation Foundation (WPF)

Bezüglich ihrer Möglichkeiten für das Erstellen eines Fensterdialogs sind sich beide Alternativen sehr ähnlich. Windows Forms war von Anfang an dabei, WPF kam später hinzu.

Ich bevorzuge in der Regel WPF, da mir der Umstand, dass der Aufbau des Fensters in XML – genauer gesagt in XAML („Extensible Application Markup Language“) – definiert wird, etwas besser gefällt. XAML wird im Microsoft-Umfeld auch in anderen Bereich eingesetzt, unter anderem für Workflow-Definitionen. Auch wenn es bei WPF weder Eingabehilfen noch einen Designer gibt (sieht man von Visual Studio ab), ist die Umsetzung vom Schwierigkeitsgrad mit Windows Forms vergleichbar. Man muss lediglich daran denken, dass es bei XML wie immer auf die Groß-/Kleinschreibung ankommt.

Um nicht jedes Mal bei null beginnen und sich irgendwo im Web ein XAML-Grundgerüst besorgen zu müssen, stelle ich im Folgenden ein Grundgerüst für ein PowerShell-Skript vor, das per WPF und XAML ein Fenster mit einem Label, einer Textbox und einem Button anzeigt. Nach einem Klick auf den Button wird der Inhalt der Textbox in einer Messagebox angezeigt und das Fenster wird geschlossen.

Das Template soll als Ausgangspunkt für eigene Dialogfelder dienen. Das erforderliche Know-how zu WPF findet man an vielen Stellen im Web, zum Beispiel http://wpftutorial.net oder http://www.wpf-tutorial.com. Auf beiden Webportalen findet man viele kleine XAML-Beispiele für die Standard-Controls, die man 1:1 in die XAML-Definition des PowerShell-Skripts übernehmen kann.

```
<#
 .Synopsis
 Template für ein WPF-Fenster
#>

$AppName = "WPFApp"
$WindowTitle = "WPF-App"
$Label1Content = "Label1"
$Button1Content = "OK"

$WindowXaml = @"
  <Window
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Title="$WindowTitle"
     FontSize = "24"
     Height="400"
     Width="600"
    >
    <DockPanel
       HorizontalAlignment = "Center"
       Margin="4,4,4,4"
       Background = "Lavender"
       LastChildFill="false"
    >
    <Label
       x:Name="Label1"
       Background = "HotPink"
       HorizontalContentAlignment = "Center"
       DockPanel.Dock = "Top"
       Content="$Label1Content"
       Height="40"
       Width="400"
       Margin="4,12,0,0"
    />
```

```
    <TextBox
       x:Name="TextBox1"
       Background = "Cyan"
       HorizontalContentAlignment = "Center"
       DockPanel.Dock = "Top"
       Text = "&lt;Irgendetwas eingeben&gt;"
       Height="40"
       Width="400"
       Margin="4,12,0,0"
  />
    <Button
       x:Name="Button1"
       Content="$Button1Content"
       DockPanel.Dock = "Bottom"
       Height="40"
       Width="200"
       Margin="4,4,0,12"
    />
  </DockPanel>
  </Window>
"@

# Die WPF-Assembly-Dateien müssen geladen werden
Add-Type -AssemblyName PresentationFramework
Add-Type -AssemblyName PresentationCore
Add-Type -AssemblyName WindowsBase

# Fenster aus dem XAML-Code anlegen
$Win = [System.Windows.Markup.XamlReader]::Parse($WindowXaml)

# Die FindName-Methode holt ein WPF-Control anhand seines Namens
$Button1 = $Win.FindName("Button1")
$Textbox1 = $Win.FindName("TextBox1")
$Textbox1Content = $Textbox1.Text

# Das Scriptblock-Element wird dem Click-Event-Handler des Button
angehängt
$Button1SB = {
    [System.Windows.MessageBox]::Show("Vielen Dank - das Eingabefeld
enthaelt: {0}" -f $Textbox1Content, $AppName)
    $Win.Close()
}

$Button1.add_Click($Button1SB)
# Fenster als modales Dialogfenster anzeigen
$Win.ShowDialog()
```

Nach dem Start des Skripts wird ein Fensterdialog angezeigt. Für das Verständnis sind drei Aspekte wichtig:

1) Jedes Bedienelement, auch Control genannt, muss per *x:Name-Attribut* einen eindeutigen Namen erhalten. Per *$Win.FindName()*-Aufruf wird dieses Control „gefunden“ und einer Variablen zugewiesen.
2) Per *add*-Methode wird der Eventhandler eingerichtet, in dem einer Control-Variablen ein Scriptblock übergeben wird. Dieser wird ausgeführt, wenn das Event eintritt.
3) Das Fenster wird per *ShowDialog()* modal angezeigt, blockiert damit während es angezeigt wird irgendwelche „Hintergrundaktivitäten“ des Skripts (Abb. 19.2).

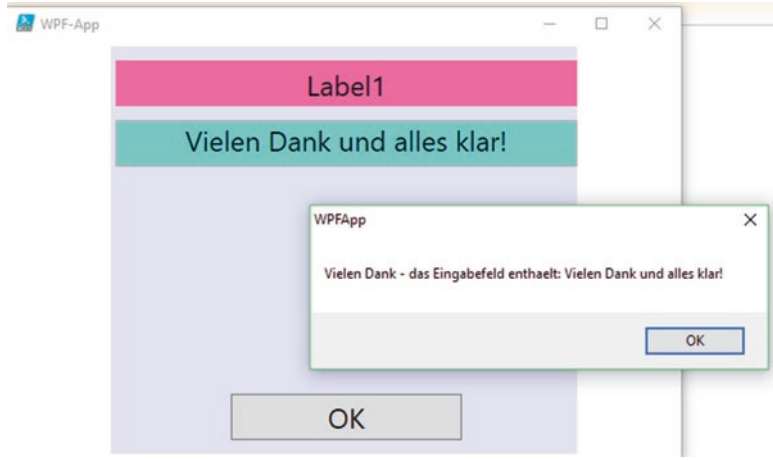

Abb. 19.2 Das WPF-Fenster wird per ShowDialog() angezeigt

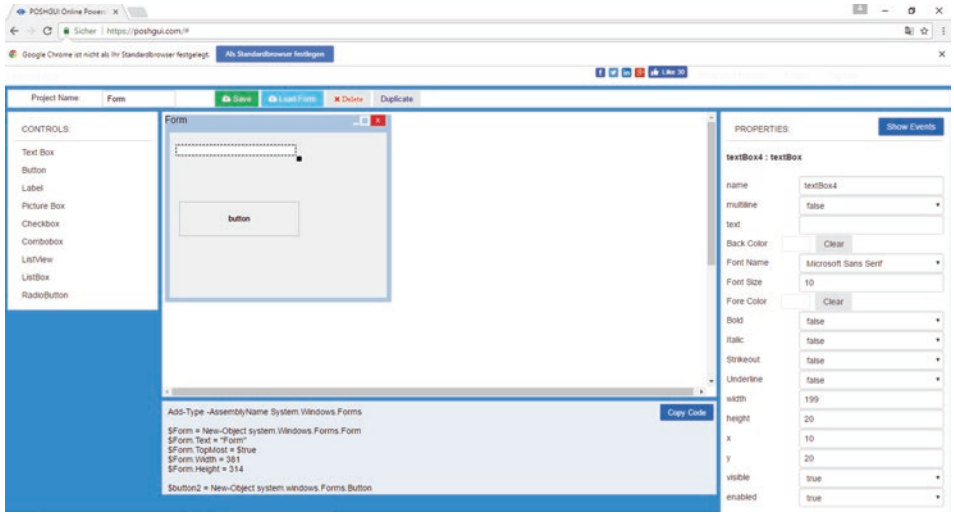

Abb. 19.3 Auf WinForms basierende Dialogfelder für PowerShell-Skripte lassen sich auch im Browser erstellen

Und: Bei XAML kommt auf die Groß-/Kleinschreibung an.

Ein Argument, das für WinForms im Vergleich zu WPF spricht ist, dass es mit *PowerShell Studio* von *Sapien Technologies* ein Tool gibt, das einen komfortablen Designer für eine WinForms-Fensteroberfläche besitzt. Damit ist ein Fensterdialog in wenigen Minuten erstellt (Abb. 19.3).

▶ **Tipp** Unter https://poshgui.com/ finden Sie einen gut gemachten browserbasierten Designer für WinForms-Oberflächen. Mit dem Zusammenstellen der Oberfläche werden die PowerShell-Befehle generiert, die Sie nur noch in ein Skript einfügen müssen, um den erstellten Dialog in einem Skript anzuzeigen. Schade, dass es so etwas nicht für WPF gibt.

19.10 Win32-API-Funktionen aufrufen

Auch der Aufruf von Funktionen des Betriebssystems, die Win32-API-Funktionen, ist per PowerShell möglich. Allerdings nicht direkt, sondern über den Umweg einer in C# geschriebenen Methodendefinition, die per *extern*-Schlüsselwort die Dll-Function aufruft.

Beispiel

Das folgende Beispiel verwendet die Win32-API-Funktionen *SendMessage* und *FindWindow*, um ein Anwendungsfenster zu schließen.

```
<#
 .Synopsis
 Schließen eines Fensters über PostMessage und WM_CLOSE
 .Description
 Wichtig: Bei FindWindow muss der erste Parameter vom Typ IntPtr sein und
es muss [IntPtr]::Zero übergeben werden
#>

$CSCode = @'
  using System;
  using System.Runtime.InteropServices;

  public static class Win32A
  {
    public static uint WM_CLOSE = 0x10;

    [DllImport("user32.dll", CharSet = CharSet.Auto, SetLastError =
false)]
    public static extern IntPtr SendMessage(IntPtr hWnd, UInt32 Msg,
IntPtr wParam, IntPtr lParam);

   [DllImport("user32.dll")]
    public static extern IntPtr FindWindow(IntPtr lpClassName, string
lpWindowName);
    }
'@

Add-Type -TypeDefinition $CSCode
```

Damit steht der Typ *Win32* zur Verfügung, der mit *FindWindow* und *SendMessage* zwei statische Methoden-Members besitzt.

Beispiel

Das folgende Beispiel geht davon aus, dass ein Anwendungsfenster mit dem Titel „Unbekannt – Paint“ geöffnet ist (*Mspaint.exe*). Es wird durch die folgende Befehlsfolge geschlossen.

```
$WinName = "Unbenannt - Paint"
$Hwnd = [Win32A]::FindWindow([IntPtr]::Zero, $WinName)
[Win32A]::SendMessage($Hwnd, [Win32A]::WM_CLOSE, 0, 0)
```

19.11 Zusammenfassung

Die PowerShell ist eine flexible Skriptsprache, die auch für Entwickler als Zweitsprache attraktiv ist. Sie basiert auf der .NET-Laufzeit und arbeitet damit mit demselben Typensystem wie etwa C#. Ein Pluspunkt ist der flexiblere Umgang mit Typen, die sich entweder per *Add-Type*-Cmdlet oder auf der Grundlage einer XML-Datei erweitern lassen. Auch das Erstellen von Assembly-Dateien ist per *Add-Type*-Cmdlet möglich. Das Zusammenspiel von WPF-Klassen mit PowerShell ist sogar etwas flexibler als unter C#. Auch für Entwickler, die unter anderem Plattformen wie Linux oder MacOSX arbeiten, ist die PowerShell inzwischen eine Alternative, da es sie auf der Basis von .NET Core auch für diese Plattformen gibt.

Glossar

.NET siehe .NET-Laufzeit

.NET Framework. siehe .NET-Laufzeit

.NET-Laufzeit Der Name einer Laufzeitumgebung von Microsoft, auf der auch die PowerShell basiert. Die beiden wichtigsten Bestandteile der .NET-Laufzeit sind die *Common Language Runtime* (CRL), eine Klassenbibliothek und Managed Code.

Alias Ein Alias ist ein Commandtyp. Ein Alias ist ein Zweitname für ein anderes Cmdlet, eine Function oder einen Verzeichnispfad. Ein Alias kann aber keine Parameterwerte umfassen. Für diese Anforderung muss eine Function definiert werden.

Assembly Ein Begriff aus dem .NET Framework. Eine Assembly ist zunächst eine logische Einheit, die Typendefinitionen und/oder Ressourcen enthält. Physikalisch liegt eine Assembly als eine Datei mit der Erweiterung *.dll* oder *.exe* vor. Die PowerShell besteht aus einer Reihe von Assembly, unter anderem *System.Management.Automation.dll*.

Attribut Ein Attribut ist in der Welt der Skript- und Programmiersprachen allgemein eine ergänzende Information für einen Typen. Bei der PowerShell können alle Parameter sowie einzelne Parameter über Attribute erweitert werden. Über Attribute wird unter anderem die Art und Weise der Parameterbindung bei einem Skript oder einer Function festgelegt. Über Attribute kann auch die Validierung für einen Parameter durchgeführt werden. Da Attribute auf Klassen basieren und sie selber Typen sind, wird ihr Name in eckige Klammern gesetzt. Da es für jedes Attribut einen Typenalias gibt, wird durch dieser Aliasname in eckige Klammern gesetzt.

Cmdlet Eine Cmdlet („Commandlet“) ist ein Command-Typ. Der Inhalt des Cmdlets besteht aus binären Befehlen, die sich nur über einen .Net-Disassembler sichtbar machen lassen. Jedes Cmdlet gehört zu einem Modul und liegt als Teil einer Assembly-Datei vor. Cmdlets werden in einer Programmiersprachen wie C# oder Visual Basic erstellt.

Container Ein Container ermöglicht die Virtualisierung einer Anwendung auf einem gemeinsamen Betriebssystem.

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2

DevOps Dies ist kein offizieller Begriff und er entstand zudem nicht im Microsoft-Umfeld. DevOps steht unter anderem für das Übertragen von agilen Prozessen, die in den letzten Jahren in der Software-Entwicklung entstanden sind, auf die Umsetzung von IT-Prozessen. Eine wichtige Änderung, die mit einer Umstellung auf DevOps-Methoden einhergeht ist, eine Bereitstellung von Ressourcen, wie einer Anwendung, in kürzen Zyklen.

Dot-sourced-Aufruf Beim dot-sourced-Aufruf eines Skripts wird durch Voranstellen eines Punktes erreicht, dass das Skript so ausgeführt wird, dass alle Functions und Variablen in den Gültigkeitsbereich der Ebene gehoben werden, in der das Skript ausgeführt wird. Auch Functions können dot-sourced aufgerufen werden.

DSC *Desired State Configuration.* DSC wurde mit WMF 4.0 eingeführt. Bei DSC werden Serverkonfigurationen textuell mit den Mitteln der PowerShell-Syntax beschrieben, in eine Mof-Datei kompiliert und entweder auf einen Remote-Computer (Node) im Push-Modus übertragen oder von diesem im Pull-Modus in regelmäßigen Abständen gezogen. Die Konfiguration wird durch den LCM auf dem Node angewendet. DSC soll nicht nur das Verteilen von Konfigurationsänderungen erleichtern, sondern auch einen „Konfigurationsdrift" vermeiden, indem Änderungen einer Konfiguration im Pull-Modus nach einem bestimmten Intervall wieder rückgängig gemacht werden, so dass der gewünschte Konfigurationszustand automatisch wieder hergestellt wird.

Function Eine Function ist ein Command-Typ. Eine Function steht für einen Scriptblock, der über den Namen angesprochen wird. Functions werden auf dem Function-Laufwerk abgelegt.

Klasse Eine Klasse fasst ein oder mehrere Members (Eigenschaften und Methoden) zusammen. Auf der Grundlage einer Klasse kann ein Objekt gebildet werden. Klassen werden über den *class*-Befehl der PowerShell definiert.

Nanoserver Nanoserver ist ein Windows Server-Kern, der nur ein Minimum an Funktionalitäten bietet und daher mit einem Minimum an Ressourcen (Arbeitsspeicher und Festplattenspeicher) auskommt. Die Hautaufgabe ist das Ausführen von Anwendungen, die keine Benutzeroberfläche besitzen, in erster Linie Webanwendungen, und das zur Verfügung stellen einzelner Server-Funktionalitäten wie File Server, DNS, DHCP, Web, Container usw. Da ein Nanoserver keine Benutzeroberfläche besitzt, kann er ausschließlich remote administriert werden. Per PowerShell oder über eine Weboberfläche. Produkttechnisch ist ein Nanoserver ein Windows Server 2016.

Runspace Ein Runspace bezeichnet eine eigene Instanz der PowerShell-Engine, inklusive Commands, Providers, Variablen, Functions und Sprachelemente. Mit der Version 5.0 wurden mit *Get-Runspace* und *Debug-Runspace* zwei Cmdlets für den direkten Umgang mit Runspaces eingeführt. Ein neuer Runspace wird über `[PowerShell]::Create()` angelegt.

Scriptblock Eine Befehlsfolge in geschweifte Klammern. Scriptblock ist bei der PowerShell ein eigener Datentyp mit dem Namen Scriptblock.

Sessionstate Der Sessionstate fasst alle „Daten“ zusammen, die in einer PowerShell-Session enthalten sind. Dazu gehören unter anderem die Werte aller Variablen. Der Sessionstate kann zum Beispiel per *$ExecutionContext.SessionState* abgefragt werden. Innerhalb einer Advanced Function steht der Sessionstate über *$PSCmdlet.Sessionstate* zur Verfügung. Im Sessionstate können beliebige Werte gespeichert werden, die im Rahmen der gesamten PowerShell-Sitzung zur Verfügung stehen. Über die Eigenschaft *LanguageMode* kann der Umfang der PowerShell-Skriptsprache eingeschränkt werden, der im Rahmen der Sitzung zur Verfügung steht.

Typ Ein Typ ist eine Definition für Elemente wie Klassen oder Konstantenlisten (Enums).

Typenalias Über einen Typenalias (auch „Type Accelerator“) genannt, kann eine Typenbezeichnung durch einen kurzen Namen abgekürzt werden. Die PowerShell definiert mehrere Dutzend Typenalias. Ein Beispiel ist *[PSCredential]*, das den Typennamen *System.Management.Automation.PSCredential* abkürzt. Eine Liste aller Typenalias liefert der folgende Befehl: *[PsObject].Assembly.GetType(„System.Management.Automation.TypeAccelerators“)::Get*. Es lassen sich auch neue Typenalias definieren.

Workflow Ein Workflow ist ein Commandtyp. Die beiden wichtigsten Unterschiede zwischen einer Function und einem Workflow sind, dass

- ein Workflow von der Workflow Engine der .NET-Laufzeit ausgeführt wird.
- ein Workflow jederzeit seinen Zustand speichern kann, so dass ein Workflow unterbrochen und zu einem späteren Zeitpunkt fortgesetzt werden kann.

Stichwortverzeichnis

P. Monadjemi, *Windows Server-Administration mit PowerShell 5.1*, X.systems.press,
DOI 10.1007/978-3-658-17666-2