Quelle: lukaszkochanek – 123RF

Neuerungen in PowerShell 7

Neue Zahl, neues Glück?

von Thorsten Butz

Vor knapp 18 Jahren erblickte die PowerShell das Licht der Welt und ist seither gereift. Vor sechs Jahren brachte Microsoft dann die erste Fassung einer neu entwickelten, plattformübergreifenden PowerShell heraus – irreführenderweise als Version 6, denn im Grunde genommen war diese erste Version mehr eine Technologiedemo. Welche interessanten Neuerungen im aktuellen 7er-Release auf Admins warten und wo die Unterschiede zum Vorgänger liegen, beleuchtet dieser Beitrag.

Microsoft steckte beim Sprung von PowerShell 5 auf 6 beziehungsweise 7 in einem schwer aufzulösenden Dilemma. Die neue Generation der Skriptsprache, ursprünglich PowerShell Core genannt, war nämlich angetreten, die Windows PowerShell möglichst reibungslos zu beerben und zugleich sich von dem Stigma der Windows Shell zu lösen – getreu dem unter Satya Nadella omnipräsenten Mantra der Plattformunabhängigkeit. Obgleich der Zusatz "Core" in der Zwischenzeit aus dem Anwendungsnamen verschwunden ist, lebt er als Kennzeichnung der PowerShell-Edition weiter.

Seitdem sind zahlreiche 6er- und 7er-Versionen als quelloffene Software unter der MIT-Lizenz mit verhältnismäßig kurzen Supportlaufzeiten erschienen, die als Anwendung parallel zur Desktopedition (Windows PowerShell) installiert werden können. Um Verwechslungen vorzubeugen, wurde das Executable der Core-Edition "pwsh" beziehungsweise "pwsh.exe" benannt, die Desktopvariante hört hingegen von Beginn an auf den Namen "powershell.exe".

Für viele unverständlich, bleibt die Desktopversion weiterhin integrierter Bestandteil aller neu erscheinenden Windows-Versionen, obgleich diese Windows PowerShell im Jahr 2016 mit der Version 5.1 ihr letztes signifikantes Update erhielt. Ein Ende dieser zwiespältigen Praxis ist nicht abzusehen, Grund genug, einmal auf die Hintergründe und Effekte dieser parallelen Entwicklung zurück- und vorauszuschauen.

.NET als Unterbau

Ende 2023 ist die PowerShell in Version 7.4 als LTS-Version erschienen und vollzieht damit den Umstieg auf .NET 8. Dieses wird als "Long Term Servicing"-(LTS)-Release seitens Microsoft für drei Jahre unterstützt. Die PowerShell adaptiert diesen Support-Lebenszyklus, der sich erheblich von den Supportlaufzeiten der verschiedenen Windows-Varianten unterscheidet. Im Kern erklärt sich hiermit, weshalb die PowerShell seit Version 6 nicht mehr integrierter Bestandteil des Betriebssystems sein kann und stattdessen als eigenständige Anwendung installiert wird. Obgleich Microsofts PowerShell-Team um Steve Lee nach wie vor federführend die Weiterentwicklung vorantreibt, ist PowerShell im engen Sinne kein Microsoft-Produkt mehr.

Von Beginn an baut die Skriptsprache wesentlich auf den Bibliotheken des .NET-Frameworks auf. Der Grund für diesen Ansatz ist verblüffend einfach: Jeffrey Snover, dem Erfinder der PowerShell, war frühzeitig klar geworden, dass es enorm viel Zeit brauchen und damit ganz erhebliche Kosten verursachen würde, eine ausreichende Anzahl attraktiver und ausgereifter Kommandozeilenwerkzeuge für

eine neue Shell von Grund auf neu zu entwickeln. So lag es nahe, die im Konzern bereits vorhandene Software als Grundlage zu nehmen und diese einem breiteren Publikum auf einfache Weise zugänglich zu machen. Zwei wesentliche Datenquellen für viele Cmdlets sind dabei die Klassen des .NET-Frameworks und die WMI-Schnittstelle. Dementsprechend sind die folgenden Befehler gleichwertig:

```
[System.Diagnostics.Process]::
  GetProcessesByName('powershell')

Get-Process -name 'Powershell'
```

Und für WMI liefern diese Kommandos identische Ergebnisse:

```
wmic /namespace:\\root\StandardCimv2
  path msft_netadapter get /value

Get-CimInstance -Namespace
  'root/StandardCimv2' -ClassName
  'MSFT_NetAdapter'`

Get-NetAdapter`
```

Die Windows PowerShell 5.1 basiert auf dem .NET Framework 4.5, dem Windows-spezifischen Vorläufer des aktuellen .NET. So ist es im Wesentlichen ein Gebot der Kompatibilität: ein Entfernen der Windows PowerShell aus dem Betriebssystem würde unmittelbar erdrutschartige Störungen beim Kunden verursachen, sodass es bei der vertrauten Windows PowerShell einschließlich des Editors PowerShell ISE und allen voneinander abhängigen Komponenten im Betriebssystem Windows bleibt.

Auf das Betriebssystem kommt es an

Um die Unterschiede zwischen den Editionen zu verstehen, lohnt sich ein Blick auf die PowerShell unter Linux oder macOS. Aufgrund zahlreicher Kniffe und Tricks, auf die wir in diesem Artikel eingehen werden, verschwimmen die Unterschiede zwischen der Version 5.1 und 7.4 unter Windows recht schnell – sehr viel deutlicher tritt der Entwicklungsstand zu Tage beim Vergleich der PowerShell auf unterschiedlichen Betriebssystemen. Die Installation der aktuellen PowerShell geht auf allen unterstützten Plattformen recht flott voran, sei es mit den gängigen Paketverwaltern oder manuell [1].

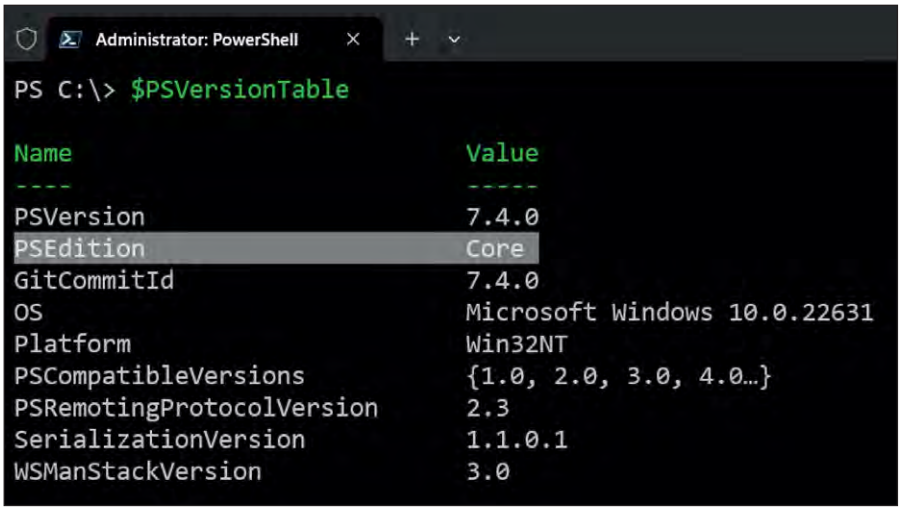

Bild 1: Der Zusatz "Core" lebt, wenn auch versteckt, weiter, wie *$PSVersionTable* zeigt.

```
## Beispielinstallation via
  HomeBrew (macOS)
brew install powershell/tap/
  powershell

## Manuelle Installation (macOS,
  Apple Silicon)
curl https://github.com/
  PowerShell/PowerShell/releases/
  download/v7.4.0/powershell-
  7.4.0-osx-arm64.pkg
  -o powershell-7.4.0-osx-arm64.pkg

sudo installer -pkg ./powershell-
  7.4.0-osx-arm64.pkg -target /
```

Installieren Sie die PowerShell 7.4 exemplarisch auf einem aktuellen macOS Sonoma mit ARM-basierter Hardware (Apple Silicon), fällt unmittelbar nach dem Start der Shell die sehr kleine Zahl an zur Verfügung stehenden Cmdlets auf. Sie finden Cmdlets zur Navigation im Dateisystem ("Get-Item", "Get-ChildItem", "Set-Location" et cetera) ebenso wie einige häufige genutzte Cmdlets ("Test-Connection", "Get-Process"). Vorsicht geboten ist ebenso bei einigen vertrauten Aliassen: "ls" oder "ps" starten die nativen Anwendungen unter macOS und Linux.

Weitere Versuche enden häufig in Frustration, es mangelt an überraschend vielen Stellen. Es gibt kein "Resolve-DNSName", "Get-NetIPConfiguration", "Get-NetAdapter", "Get-NetRoute". Ebenso fehlen Cmdlets zur Benutzerverwaltung, darunter "Get-LocalUser" oder "Get-LocalGroup". Laienhaft betrachtet kann dies insofern überraschen, als dass es ja nun fraglos auf allen unterstützten Betriebssystemen einen TCP/IP-Stack gibt, den Admins mit den erwähnten Cmdlets verwalten möchten. Ebenso kennen macOS, Linux und Windows lokale Benutzer und Gruppen. Weniger überraschend ist es da, dass die Suche nach WMI/CIM-Cmdlets vergebens ist, da die korrespondierende Schnittstelle unter macOS und Linux (weitgehend) unbekannt ist.

In der Summe bleiben in dem Beispielszenario weniger als 300 Cmdlets und nur zehn Module "built-in". Egal, welches Betriebssystem Sie nutzen: Abseits von Windows ist die PowerShell auf einen von zwei wesentlichen Bestandteilen reduziert, nämlich die Shell.

An dieser Stelle gilt es, ein gängiges Missverständnis aufzuklären: Obgleich der Name etwas anderes suggeriert, ist die PowerShell viel mehr als "nur" eine simple Shell. Sie ist ein komplexes Automatisierungs-Framework mit einem Kommandozeileninterpreter (der namensgebenden "Shell") und einer eigenständigen Skriptsprache. Die in Windows verfügbaren Cmdlets sind in vielen Fällen Teil des Betriebssystems und werden im Prinzip unabhängig von der PowerShell weiterentwickelt. Betrachten wir als Beispiel das Cmdlet "Get-Localuser":

Windows PowerShell (alias Desktop-Edition)	PowerShell (alias Core-Edition)
$home\Documents\WindowsPowerShell\Modules	$home\Documents\PowerShell\Modules
	C:\Program Files\PowerShell\Modules
	C:\Program Files\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules	C:\Program Files\PowerShell\7\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules	C:\Windows\system32\WindowsPowerShell\v1.0\Modules

Bild 2: Das Ausgeben der Umgebungsvariablen mit *$env:PSModulePath*.

```
## Microsoft.PowerShell.
  LocalAccounts

Get-Command -Name Get-LocalUser
Get-Command -Module
  Microsoft.PowerShell.LocalAccounts

Get-Module -ListAvailable -Name  Mi-
  crosoft.PowerShell.LocalAccounts |
  Select-Object -Property ModuleBase
```

Wenn Sie die genannten Befehle ausführen, erkennen Sie, dass die Quelle für den gesuchten Befehl und dessen Geschwister-Cmdlets unterhalb der Dateistruktur "c:\windows" abgelegt ist. Zu Ende gedacht wären in einer idealen Welt also Apple und die diversen Linux-Distributoren die Urheber für die jeweils plattformspezifischen Befehle zur (lokalen) Benutzerverwaltung. Im Kern ist dies ja auch so: Sie können Benuter mit "dscl" (macOS) oder "adduser" (Linux) anlegen. Nur, dass es sich hierbei halt nicht um intuitiv benutzbare Cmdlets handelt.

Verschlungene Pfade

Installieren Sie die PowerShell 7.4 auf einem aktuellen Windows 11, finden Sie zunächst die von der Windows PowerShell gewohnt hohe Zahl an Cmdlets, auch "Get-LocalUser" ist dabei. Die PowerShell 7 verwendet hierfür unter Windows einen simplen Trick, der sichtbar wird, wenn Sie sich die Umgebungsvariable "PSModulePath" ausgeben lassen (Bild 2).

Die PowerShell 7 durchsucht die systemweit verfügbaren Erweiterungen der Windows PowerShell. Würden wir diese Pfade entfernen, liegt die Zahl der verfügbarbaren Cmdlets auf dem Niveau von macOS und Linux. So aber stehen bis auf wenige Ausnahmen, wie zum Beispiel dem Cmdlet "Get-ControlPanelItem", dieselben Befehle zur Verfügung. Die nativen PowerShell-7-Cmdlets ermitteln Sie dabei mit

```
$modules = Get-Module -ListAvailable
  | Where-Object -FilterScript {
  $_.path -like "$pshome*" }

Get-Command | Where-Object
  -FilterScript { $_.Source -in
     $modules.name } |

Sort-Object -Property Name |
  Get-Unique | Measure-Object
```

Die Standard-Suchpfade erzeugen aber noch einen weiteren, recht aberwitzigen Effekt: Installieren Sie doch einmal versuchsweise ein Modul aus der PowerShell-Gallery in der PowerShell 7:

```
Find-Module -Name ImportExcel |
  Install-Module

Get-Module -ListAvailable -Name
  ImportExcel | Select-Object
  -Property ModuleBase
```

Sie werden das neue Modul im Ordner "$home\Documents\PowerShell\Modules\ImportExcel\<Version>" finden, wenn Sie es mit PowerShell 7 nachrüsten. Diese installiert von Haus aus im UserScope. Selbst wenn Sie mittels *Install-Module -Scope AllUsers* systemweit installiert hätten, wäre die Windows PowerShell in jedem Fall blind für das neue Modul gewesen.

Das muss an sich nicht schlecht sein, irritierend ist aber in jedem Fall, dass in umgekehrter Reihenfolge – Sie installieren also mit dem exakt selben Befehl aus dem Listing oben – die Erweiterung in der Windows PowerShell "ImportExcel" unmittelbar in beiden Generationen zur Verfügung steht. Denn die Windows PowerShell installiert im Standard systemweit im Pfad "C:\Program Files\WindowsPowerShell\Modules\", also einem der beiden Pfade, die auch die PowerShell 7 durchsucht.

Natürlich lässt sich dieses Verhalten jederzeit abändern: Sie können die Umgebungsvariable anpassen oder schlicht alle Module im UserScope abspeichern, um eine saubere Trennung zu erreichen; für den Admin ist dieses Verhalten jedoch schlicht inkonsistent und irritierend. Er oder sie wird zu dem Schluss kommen, dass Module am besten mithilfe der Windows PowerShell bereitgestellt werden. Problem vermeintlich gelöst, aber auch das Erstellen eigener Profile erzeugt Verwirrung.

In Bild 3 ist unschwer zu erkennen, dass es keine Übereinstimmung bei den Profilpfaden gibt. Grundsätzlich sinnvoll, aber wer will schon zahlreiche unterschiedliche Profile pflegen. Insbesondere dann, wenn Einstellungen in beiden Editionen in gleicher Weise funktionieren. Ein simpler und flexibler Ansatz besteht darin, ein gemeinsames Profil an beliebigen Stellen zu hinterlegen – das kann ein OneDrive-Ordner oder sogar ein Github-Repo sein – und die Profildateien auf das Laden dieses Profils zu beschränken. Sie laden ein gemeinsames Profil mittels "dot sourcing" wie folgt:

```
. .\$home\Documents\myprofile.ps1
```

Alternativ können Sie Ihr Benutzerprofil um einen editionsübergreifenden Ordner bereichern, in dem Sie eine Verzeichnisverbindung erstellen, die von PowerShell zur Windows PowerShell zeigt. Auf diese Weise haben Sie unmittelbar ein gemeinsam nutzbares Profil und ebenso einen geteilten Modulordner. Die Umleitung mittels Verzeichnisverbindung im Dateisystem erfolgt über

```
New-Item -ItemType Junction -path
  "$home\Documents\PowerShell"
  -Value "$home\Documents\
     WindowsPowerShell"
```

Es gibt sie noch, die guten Cmdlets

Sie können das Einbinden der eigenen Module in Windows PowerShell als

$profile	Windows PowerShell	PowerShell
AllUsersAllHosts	C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1	C:\Program Files\PowerShell\7\profile.ps1
AllUsersCurrentHosts	C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1	C:\Program Files\PowerShell\7\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts	$home\Documents\WindowsPowerShell\profile.ps1	$home\Documents\PowerShell\profile.ps1
CurrentUserCurrentHost	$home\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1	$home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1

Bild 3: In Sachen Profile gibt es Unterschiede zwischen der Windows PowerShell und der PowerShell.

Workaround betrachten, der PowerShell 7 im Vergleich zu seinem Urahn mindestens ebenbürtig erscheinen lässt. Darin ließe sich aber auch ein wesentlicher Grund für die stotternde Weiterentwicklung des Ökosystems erkennen. Schließlich sind jene Module, die als Bestandteil von Windows ausgeliefert werden, von den rasanten Releasezyklen der PowerShell ausgenommen. Nichtsdestotrotz haben eine Reihe von Cmdlets über die Jahre nützliche Verbesserungen erfahren. Lassen Sie uns exemplarisch einen Blick auf solche Verbesserungen werfen.

Ihnen dürfte in Bild 4 auffallen, dass die Datei "computers.txt" in unserem Beispiel etwas unglücklich aufgebaut ist. Da die Computernamen in einer Zeile stehen, wird "Get-Content" den Inhalt der Datei als einzelne Zeichenfolge interpretieren. Der Parameter "-delimiter" erzeugt stattdessen ein Array mit drei Elementen, die an "Test-Connection" übergeben werden. So könnten wir wahlweise in PowerShell 7 anstelle des ICMP echo requests einen einfachen TCP-Portscan initialisieren:

```
Get-Content -Path .\computers.txt
  -Delimiter ',' | Test-Connection
  -Count 1
Get-Content -Path .\computers.txt
  -Delimiter ',' | Test-Connection
  -TcpPort 3389
```

In der Windows PowerShell scheitern wir mit dem oben gezeigten Code jedoch mehrfach:

- Der Parameter "-delimiter" belässt sinnloserweise das Trennzeichen in der Ausgabe.
- "Test-Connection" unterstützt kein "Pipelining by value".
- "Test-Connection" kennt keinen Parameter "-TcpPort".

Die Aufgaben des Listings lassen sich jedoch auch mit der Windows PowerShell lösen, wenn Sie den Code wie hier ein wenig umformen:

```
Test-Connection -ComputerName
  (Get-Content -Path
  'computers.txt').split(',')
    -Count 1
(Get-Content -Path
  'computers.txt').split(',') |
  Test-NetConnection -Port 3389
```

Zugegeben, das Windows PowerShell-konforme Listing ist kein "schöner Code", aber schöner Code ist im Admin-Alltag ein schwaches Argument. Entscheidend ist, dass diese zweite Variante in beiden Editionen funktioniert.

Zu den auffälligsten Neuerungen der Version 7 gehören eine Reihe von Optimierungen, die die (lokale) Verarbeitung spürbar beschleunigen können. Die Urgesteine "Group-Object" und "Sort-Object" arbeiten deutlich effizienter und "ForEach-Object" kann nun Skriptblöcke unter Zuhilfenahme multipler Threads parallelisieren. Ein Beispiel zum Multithreading in PowerShell 7:

```
1..10 | ForEach-Object -Parallel {
  Start-Sleep -Seconds 1 }
```

Ihnen wird unmittelbar auffallen, dass die genannte Anweisung keinesfalls zehn Sekunden in Anspruch nimmt, stattdessen erfolgt die Ausführung parallel. Im Hintergrund arbeitet das neu entwickelte "ThreadJobs"-Modul, das sich an die Syntax der Job-Cmdlets anlehnt:

```
Start-ThreadJob -ScriptBlock {
  Start-Sleep -Seconds 1
  'Finished!'
}
Get-Job | Wait-Job | Receive-Job
```

Wenn Sie den ersten Befehl des letzten Listings mehrfach aufrufen, imitieren Sie im Prinzip das Verhalten des parallelisierten "ForEach-Object". Interessanterweise können Sie das ThreadJob-Modul ohne Weiteres auch in der Windows PowerShell laden:

```
Import-Module -Name 'C:\Program
  Files\PowerShell\7\Modules\
    ThreadJob'
```

Hierbei zeigt sich einmal mehr ein fast schon tragischer Aspekt der zwei PowerShell-Editionen: Obgleich es technisch geradezu trivial erscheint, "ForEach-Object -parallel" auch in der Windows PowerShell anzubieten, werden dort prinzipiell keine Neuerungen implementiert.

Eine der praktischsten Verbesserungen der PowerShell 7 ist übrigens der ConciseView (concise = prägnant) für Fehlermeldungen, der im Standard die Ausgabe von Fehlern auf das Wesentliche reduziert, analog zu "$Error[0].Exception.Message". Das neue Cmdlet "Get-Error" liefert ergänzend eine sehr detaillierte Ansicht zum (letzten) Fehler.

Remoting nutzen

PowerShell unterstützt Fernaufrufe mittels SSH und WinRM/WSMan. Mit der Implementierung der SSH-Technologie bedient sich die PowerShell nun endlich auch der Lingua Franca für Fernverbindungen:

```
Invoke-Command -HostName lin-sv1
  -UserName Linus
```

Obgleich SSH-Server und -Clients für Windows verfügbar sind, nutzt Windows

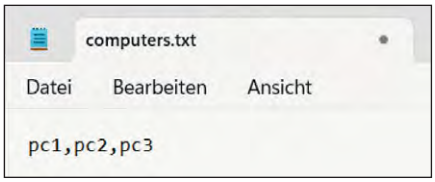

Bild 4: Die Datei "computers.txt" ist unglücklich formatiert.

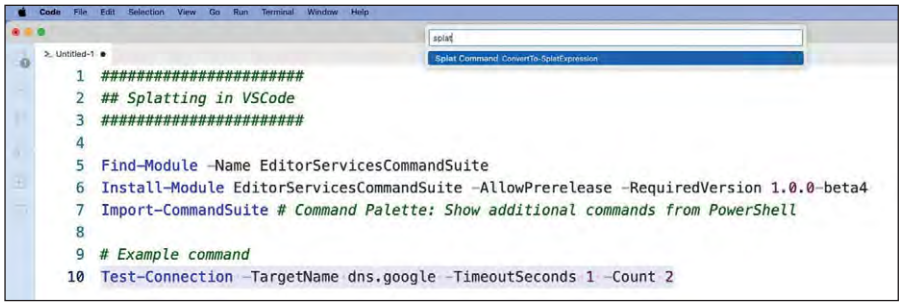

Bild 5: Das Splatting in VSCode vorher…

```
9  # Example command
10 $testConnectionSplat = @{
11     TargetName = 'dns.google'
12     TimeoutSeconds = 1
13     Count = 2
14 }
15
16 Test-Connection @testConnectionSplat
```

Bild 6: … und hinterher.

im Standard weiterhin die vertraute WinRM-Kommunikation. Ein SSH-Dienst muss unter Windows zunächst nachgerüstet werden, während WinRM auf Windows Server standardmäßig aktiv ist.

Gesetzt den Fall, dass Sie die PowerShell nur auf Ihrem Client nachinstallieren, können Sie problemlos eine Remoteverbindung mit einem entfernten Windows-Rechner aufbauen – auch wenn dort "nur" die integrierte Windows PowerShell läuft:

```
$cred = Get-Credential
Test-WSMan -ComputerName win-dc1
  -Authentication Kerberos
  -Credential $cred
$PSSession = New-PSSession
  -ComputerName win-dc1
  -Credential $cred

## Fan out
Invoke-Command -Session $PSSession
  -ScriptBlock {
    hostname
    whoami
    $PSVersionTable.PSVersion.
        ToString()
}

## Implicit remoting
$proxyCmdlets = Import-PSSession
  -Session $pssession
  -Module ActiveDirectory
Get-ADDomain

## Exportieren des Proxy-Moduls
Export-PSSession -OutputModule
  ActiveDirectory -Session
  $PSSession -Module ActiveDirectory
  -AllowClobber
```

Eben jenes "implicit remoting", das wir im letzten Beispiel bewusst eingesetzt haben, verwendet die PowerShell im Hintergrund, um einige Cmdlets zugänglich zu machen, die nicht kompatibel zur Core-Edition sind. Ein auffälliges Beispiel ist "Get-EventLog". Rufen Sie den Befehl einmal in der PowerShell auf und schauen Sie ganz genau hin, erkennen Sie, dass die PowerShell eine Remoteverbindung zum eigenen Rechner startet und die Windows PowerShell um Ausführung des Befehls bittet. Das ist zugegebenermaßen ein Taschenspielertrick, da es für diese Art der Ausführung nicht einmal besonderer Benutzerberechtigungen bedarf. Es führt aber erwartbar auch zu kleineren oder größeren Problemen, was sich beispielhaft am Cmdlet "New-LocalUser" zeigt, dessen Geschwister-Cmdlet "Get-LocalUser" schon einige Male als Beispiel Verwendung fand:

```
New-LocalUser -Name 'Jeffrey'
  -NoPassword

New-LocalUser: Could not load type
  'Microsoft.PowerShell.Telemetry.
  Internal.TelemetryAPI' from assem-
  bly 'System.Management.Automation,
  Version=7.4.0.500, Culture=
  neutral, PublicKeyToken=
  31bf3856ad364e35'.
```

So liegt es nahe, dass sich dieses Verhalten explizit unterbinden lässt und den Rückgriff auf die Windows PowerShell auf diesem Weg verhindert [2].

Besser Formatieren mit VSCode

Pragmatisch betrachtet spielen die oft filigranen Unterschiede zwischen den PowerShell-Editionen im Alltag kaum eine Rolle. Im Kontext selbstverwalteter lokaler Windows-Netzwerke ist Kompatibilität ein so hohes Gut, dass kaum ein Skript die Neuerungen der PowerShell 7 adressiert. Im Bereich der Clouddienste jedoch ändert sich das Bild. Im Schlepptau der PowerShell haben sich neue Werkzeuge wie das Windows Terminal etabliert, vor allem aber Visual Studio Code (VSCode), das Sie übrigens wie folgt installieren:

```
$uri = 'https://update.code.
  visualstudio.com/latest/
  win32-x64/stable'
Invoke-WebRequest -UseBasicParsing
  -Uri $uri -OutFile 'VSCodeSetup-
  x64.exe'
$args = '/verysilent /tasks=
  addcontextmenufiles,
  addcontextmenufolders,addtopath'
Start-Process -FilePath
  'VSCodeSetup-x64.exe'
  -ArgumentList $args -Wait
```

VSCode ist eigentlich kein perfekter Begleiter für die PowerShell – basierend auf dem Electron-Framework, geschrieben in TypeScript und mit der PowerShell über eine notorisch instabile "work-in-progress"-Erweiterung verheiratet. Ein Editor für "al-

Listing 1: Konfigurationsdatei

```
"C:\Program Files\PowerShell\7\
   powershell.config.json"

{
  "Microsoft.PowerShell:ExecutionPolicy":
   "Unrestricted",
  "WindowsPowerShellCompatibilityModule-
       DenyList": [
    "PSScheduledJob",
    "BestPractices",
    "UpdateServices"
  ],
  "DisableImplicitWinCompat": true  //
   disable WinPSCompatSessions
}
```

les", kein passgenaues Werkzeug für eine bestimmte Anwendung. Doch hat sich rund um VS- Code und Plattformen wie GitHub in den letzten Jahren ein beispielloses Ökosystem entwickelt, sodass durch Source Control und eine stetig wachsende Anzahl interessanter VSCode-Erweiterungen ein großes Potenzial erwächst.

Nach der Installation der originären PowerShell-Erweiterung für VSCode steht Ihnen unmittelbar ein Formatter zur Verfügung (rechte Maustaste: "Format Document"), der auf Wunsch Pseudoynme durch deren vollständigen Cmdlet-Namen ersetzt. Die in den Bildern 5 und 6 beispielhaft gezeigte Erweiterung ergänzt VSCode auf Wunsch um automatisches Splatting – eine enorme Arbeitserleichterung und ein wesentlicher Schritt in Richtung lesbaren Code ("posh code"). In der Konfigurationsdatei "settings.json" lassen sich eine Reihe nützlicher Anpassungen vornehmen. Dabei ist der bekannte ISE-Mode (Command-Palette / PowerShell / Enable ISE Mode) nur der erste Schritt. Im folgenden Listing mit dem Inhalt der Datei "settings.json" finden Sie Ideen, die Sie testen sollten:

```
{
    // ISE compatible encoding
    "[powershell]": {
        "files.encoding": "utf8bom"
    },

    // Editor settings
    "editor.formatOnSave": true,
    "editor.formatOnSaveMode":
       "modifications",
    "powershell.codeFormatting.
       autoCorrectAliases": true,

    // Disable "annyoing" snippets
        and suggestions
    "editor.snippetSuggestions":
        "button",    // "none"
        disables these suggestions
    "editor.parameterHints.enabled":
        false,
    "powershell.scriptAnalysis.
        enable": false //
        distraction free mode
}
```

Der PSScriptAnalyzer (letzte Zeile der Konfiguration im Listing oben) ist ein ausgesprochen sinnvolles und nützliches Werkzeug. Dennoch hilft es manchmal, die klugen Ratschläge temporär auszublenden. Ebenso möchten Sie unter Umständen die recht aufdringliche Autovervollständigung der PowerShell-Erweiterung ein wenig zügeln. Die von der ISE gewohnten Grundfunktionen zur Autovervollständigung von Cmdlets, Parametern und Argumenten bleiben Ihnen auch in dieser Konfiguraton erhalten.

Die Bedürfnisse und Erwartungen an die Technologie der PowerShell hängen ganz maßgeblich von der Jobrolle, den eigenen Aufgaben und den zugehörigen Produkten ab. Wer sich auf den Weg macht, Microsofts stetig wachsendes Abonnement-basiertes Serviceangebot zu nutzen (alias Cloud Computing), sollte zwingend einen Blick auf die "Azure Cloud Shell" werfen. Unter Windows führt der schnellste Weg über das Windows Terminal, die Cloud Shell lässt sich aber auch im Konsolenbereich direkt in VSCode einbinden [3].

Der zentrale Vorteil: In einem vorkonfigurierten Container finden sich zahlreiche Module und ergänzende Werkzeuge. Sie können so unmittelbar auf Azure- und Microsoft 365-Ressourcen zugreifen, ohne die Erweiterungen installieren oder aktualisieren zu müssen. Und damit ist es mehr als ein Bonmot, dass die Azure Cloud Shell auf Linux basiert und mittels Dockercontainer bereitgestellt wird.

Fazit

VSCode und dessen Ökosystem sind reizvoll und interessant – doch die Windows PowerShell 5.1 wird dort ebenfalls unterstützt. Für einen Admin also kein Grund, auf die neue Generation der Skriptsprache zu wechseln. Gleiches gilt für das Windows-Terminal: eine sehr willkommene Verbesserung für die angestaubte Conhost-Welt, aber ebenso für beide Editionen verfügbar. Selbst die auf den ersten Blick auffälligste Neuerung der PowerShell 7.4 entpuppt sich als editionsübergreifendes Feature. So bietet das aktualisierte PSReadline-Modul eine erweiterte Autovervollständigung auf Grundlage einer sitzungsunabhängigen Historie an. Ein Update genügt, um diese Funktion auch in der Windows PowerShell zu aktivieren. Gleiches gilt für die runderneuerte PowerShell-Paketverwaltung "PowerShellGet".

Außerhalb ihres angestammten Terrains tut sich die PowerShell allerdings schwer. Linux- und macOS-affine Anwender fremdeln mit der objektorientierten Shell aus dem Hause Microsoft. Zu dominant sind Python und Co., zu spärlich finden sich Anwendungsfälle außerhalb der Windows/Azure-Welt, die unmittelbar nützlich sind und den Neuling begeistern. Ausgerechnet das etwas sperrige VSCode ist hier ein Türöffner. So gibt es häufig Auszubildende und junge Admins, die leidenschaftliche VSCode-Nutzer sind und mit einem neuen Bild von Microsoft aufwachsen. Diese sind durchaus begeisterungsfähig für die PowerShell. Der traditionell eher kommandozeilen-agnostische Windows-Admin tut sich schwer. Praktische Vorteile sind on-premises kaum zu erkennen, die PowerShell ISE ist nach wie vor ein flottes und stabiles Tool.

Am Ende verdienen Webservices als Schnittstelle zwischen alter und neuer Welt unsere Beachtung. REST-APIs sind wichtig und werden immer bedeutsamer. Und nirgends spielt die PowerShell so elegant ihre Stärken aus. Nach zahllosen, mäßig gelungenen Versuchen, spezifische Module für Azure und Co. anzubieten, stellt Microsoft aktuell die Verwaltungsschnittstelle des Cloudangebots auf die Graph-API um; eine REST-API als Metaschnittstelle für unterschiedlichste Dienste. Viele populäre Anwendungen wie ServiceNow, aber auch Fachverfahren bieten oftmals ebensolche webbasierten Schnittstellen. So lohnt ein zweiter Blick auf die Cmdlets "Invoke-WebRequest" und "Invoke-RestMethod", die in PowerShell 6 und 7 maßgeblich erweitert wurden. *(dr)* IT

Link-Codes

[1] Installation der PowerShell
os11h

[2] Windows-PowerShell-Kompatibilität
os11i

[3] Cloud Shell in VSCode einbinden
os11j