Was die Frage aufwirft: Was genau tut `Foreach-Object` hier eigentlich? Das führt uns direkt zum nächsten Abschnitt:

Befehle verbinden

Gäbe es für jedes Problem der Welt den einen passenden Befehl, wäre Automation sehr einfach. Dann allerdings müsste PowerShell aus Millionen von Befehlen bestehen, und selbst dann würde Ihnen sicher noch eine neue Aufgabe einfallen, die davon nicht abgedeckt ist.

Deshalb geht PowerShell einen anderen Weg und macht seine Befehle teamfähig. Im Team können einzelne Befehle so gut wie jedes noch so komplexe Problem lösen. Stellen Sie sich deshalb die Befehle am besten vor wie Legobausteine, aus denen Sie die unterschiedlichsten Automationslösungen zusammenbauen können.

Im einfachsten Fall besteht die Teamarbeit bloß aus mehreren voneinander unabhängigen Einzelbefehlen, die wie beim Akkureport in Listing 2.3 der Reihe nach abgearbeitet werden. Viel häufiger ist es aber nötig, dass Befehle als echtes Team zusammenarbeiten müssen, indem ein Befehl die Ergebnisse eines anderen Befehls weiterverarbeitet.

Das Türchen-Modell

Wenn ein Befehl Ergebnisse zurückliefert, implementiert PowerShell drei »Türchen«, durch die die Ergebnisse des Befehls geleitet werden können (siehe Abbildung 2.16). Eines der drei Türchen wird immer verwendet. Dabei geht PowerShell in der in Abbildung 2.16 gekennzeichneten Reihenfolge vor. Wenn Sie keins der Türchen einsetzen, nutzen Befehlsergebnisse automatisch »Türchen 3«.

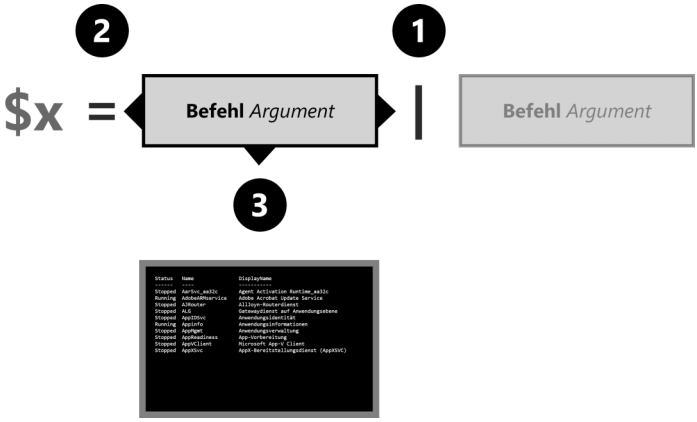

Abbildung 2.16: Weiterleiten von Ergebnissen über drei verschiedene »Türchen«

Wir schauen uns als Nächstes an, was bei den drei Türchen genau geschieht, und beginnen mit »Türchen 3«. Das kommt nämlich vollautomatisch zum Einsatz, wenn Sie kein besonderes Türchen wählen.

Normalfall: »Türchen 3«

»Türchen 3« wandelt die Ergebnisse des Befehls in Text um und gibt diesen Text sichtbar in der PowerShell-Konsole aus. Das erklärt schon einmal, warum Sie bei PowerShell zur Ausgabe keine besonderen Ausgabebefehle wie `echo` oder `print` benötigen:

```
PS> Get-Date

Donnerstag, 22. Oktober 2020 11:31:10
```

Auch wenn Sie Berechnungen durchführen, landet das Ergebnis dank »Türchen 3« ohne weitere Ausgabebefehle sichtbar in der Konsole:

```
PS> 100 * 2 / 44 + 8.65
13,1954545454545
```

Wenn Sie genau hinschauen, entdecken Sie die Spuren von »Türchen 3« und seiner Textumwandlung: Warum war wohl das Dezimaltrennzeichen in der Befehlszeile noch ein Punkt (.), im Ergebnis dagegen ein Komma (,)? Ein Blick hinter die Kulissen erklärt das:

- **Textumwandlung:** Konsolen können ausschließlich Text anzeigen. Die Ergebnisse der Befehle werden also immer in Text umgewandelt. Das, was Sie in der Konsole sehen, ist demnach nur eine Annäherung an die wahren Daten, denn die allermeisten Befehle liefern keinen Text zurück, sondern Informationen ohne feste Gestalt (sogenannte *Objekte*). Den Unterschied werden Sie gleich noch genauer sehen können.

  Nur bei der Umwandlung in Text wird – wo anwendbar – die aktuelle Spracheinstellung Ihres Betriebssystems berücksichtigt. Deshalb erscheint der Wochentag in obigem Beispiel in Deutsch als »Donnerstag«, und bei Zahlen wird das Dezimaltrennzeichen verwendet, das in den regionalen Spracheinstellungen Ihres Betriebssystems vermerkt ist (auf deutschen Systemen ein Komma). Auf einem englischen oder französischen System sähen die Ausgaben ganz anders aus. Hier die Ausgabe der Befehle von eben auf einem englischen System:

  ```
  PS> Get-Date

  Thursday, October 22, 2020 11:33:12 AM

  PS> 100 * 2 / 44 + 8.65
  13.1954545454545
  ```

- **Informationsverlust:** Der Platz in der Konsole ist begrenzt. Die Ergebnisse eines Befehls werden (wenn nötig) gekürzt. In der Textausgabe der Konsole können Informationen verloren gehen, Spalten werden möglicherweise nicht vollständig angezeigt oder mit »...« abgekürzt.

»Türchen 3«, die Ausgabe in die Konsole, ist also nur eine Vorschau, mit der Sie Befehlsergebnisse näherungsweise begutachten können. Für präzise und reproduzierbare Automation eignet sich die auf Text reduzierte Information in der Konsole hingegen nicht mehr.

Dafür sind die anderen beiden Türchen gedacht. Als Nächstes schauen wir uns an, wie die Ergebnisse von einem Befehl direkt zu einem anderen »streamen« können, beinahe wie eine Eimerkette bei der Feuerwehr.

Modernes Pipeline-Streaming: »Türchen 1«

Mit dem Pipeline-Operator (|) werden Befehle direkt miteinander gekoppelt: Die Ergebnisse des ersten Befehls werden direkt an den folgenden Befehl weitergereicht und dabei nicht verändert, umgewandelt oder gekürzt.

In der Konsole wurde die Datumsinformation von `Get-Date` zum Beispiel in die Sprache des Betriebssystems übersetzt und als Datumsangabe formatiert:

```
PS> Get-Date

Donnerstag, 22. Oktober 2020 11:31:10
```

Wenn Sie das Ergebnis von `Get-Date` dagegen mit dem Pipeline-Operator | weiterleiten, erhält der nachfolgende Befehl das unveränderte Originalergebnis von `Get-Date`:

```
PS> Get-Date | Out-GridView
```

`Out-GridView` zeigt alles, was man ihm sendet, in einem separaten Fenster an (siehe Abbildung 2.17).

Hinweis

`Out-GridView` steht zwar nur unter Windows zur Verfügung, es soll hier aber auch bloß stellvertretend für beliebige Cmdlets die Datenübertragung per Pipeline veranschaulichen. Selbst wenn Sie `Out-GridView` auf Ihrem Betriebssystem also nicht ausführen können, finden sich die entscheidenden Details direkt in Abbildung 2.17.

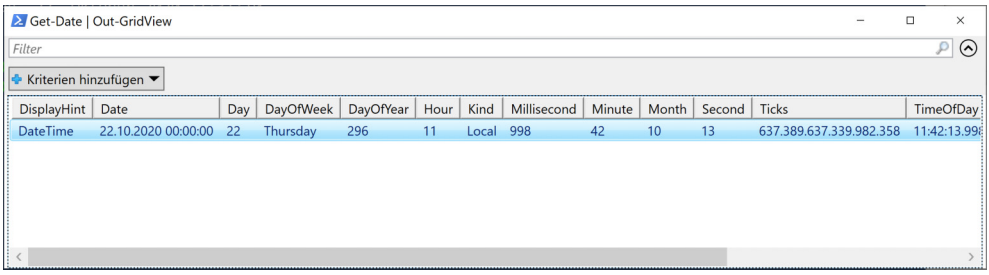

Abbildung 2.17: In der Pipeline bleiben Daten strukturierte Objekte.

`Out-GridView` offenbart: `Get-Date` liefert in Wirklichkeit ein *Datumsobjekt*, das die Zeitinformation in vielen separaten Spalten (sogenannten *Eigenschaften*) mit allen wichtigen Details (und stets in Englisch) anzeigt. Mit diesen Informationen kann ein Automationsskript verlässlich arbeiten. Sie werden in einem Moment sehen, wie Sie auf die einzelnen Informationen des Datumsobjekts konkret zugreifen. Zuerst schauen wir uns aber die wichtigsten Pipeline-Cmdlets an.

Zwar kann man grundsätzlich beliebige Cmdlets mithilfe der Pipeline koppeln, aber in beinahe jedem PowerShell-Skript findet sich eine Handvoll Cmdlets, die mithilfe der Pipeline Befehlsergebnisse aufbereiten, auswählen und filtern können (die ausführliche Darstellung finden Sie später in Kapitel 5). Diese besonderen Pipeline-Cmdlets schauen wir uns jetzt etwas genauer an, bevor wir dann zum Datumsobjekt von eben zurückkehren.

PowerShell-SQL

Oft liefern Befehle zwar prinzipiell die richtigen Informationen, aber perfekt sind sie noch nicht. Vielleicht würden Sie gern bestimmte Spalten ausblenden (oder auch andere einblenden) oder einige Datensätze herausfiltern.

Genau dafür ist eine Gruppe bestehend aus sechs Cmdlets zuständig, die die ehrwürdige Datenbankabfragesprache *SQL* adaptiert. SQL hat sich bei Datenbanken über Jahrzehnte bewährt, und auch bei Datenbanken geht es ja darum, sich gezielt bestimmte Informationen aus den Gesamtdaten auszuwählen. So hat man dieses Konzept einfach auf die PowerShell übertragen und mit insgesamt sechs Cmdlets abgebildet.

Die beiden wichtigsten Cmdlets heißen `Select-Object` (Spalten aussuchen) und `Where-Object` (Zeilen aussuchen).

Erwünschte Spalten selbst festlegen

Mit `Select-Object` bestimmen Sie, welche Spalten (Eigenschaften) für Sie wichtig sind. In Abbildung 2.17 haben Sie gesehen, dass `Get-Date` in Wirklichkeit das Datum in viele Einzelinformationen aufsplittet. Mit `Select-Object` könnten Sie sich daraus die Informationen herauspicken, die Sie benötigen:

```
PS> Get-Date | Select-Object -Property Day, DayOfWeek, DayOfYear, Year

Day DayOfWeek DayOfYear Year
--- --------- --------- ----
 28    Monday       363 2020
```

Tipp

`Select-Object` unterstützt Platzhalterzeichen (*). Sie können also auch sämtliche Eigenschaften sichtbar machen – oder alle Eigenschaften anzeigen, die ein bestimmtes Wort enthalten:

```
PS> Get-Date | Select-Object -Property *Day*

Day DayOfWeek DayOfYear TimeOfDay
--- --------- --------- ---------
 28    Monday       363 11:07:35.3745108

PS> Get-Date | Select-Object -Property *

DisplayHint : DateTime
DateTime    : Montag, 28. Dezember 2020 11:07:41
Date        : 28.12.2020 00:00:00
Day         : 28
DayOfWeek   : Monday
DayOfYear   : 363
Hour        : 11
Kind        : Local
Millisecond : 13
Minute      : 7
Month       : 12
Second      : 41
Ticks       : 637447504610134415
TimeOfDay   : 11:07:41.0134415
Year        : 2020
```

Nebenbei wird ein weiterer Automatismus der PowerShell sichtbar: Normalerweise werden Ergebnisse als Tabelle formatiert. Würde die Tabelle aber zu breit, formatiert PowerShell die Ergebnisse automatisch als Liste. Wenn Sie mit `Select-Object` beispielsweise mehr als vier Spalten auswählen, ist das Ergebnis immer eine Liste.

`Select-Object` kann auf sämtliche Eigenschaften der Objekte zugreifen, also auch auf jene, die aufgrund von Platzmangel oder der Textumwandlung in der Konsole normalerweise nicht sichtbar sind (siehe Abbildung 2.16). `Get-Process` liefert zum Beispiel Informationen, die nur sichtbar werden, wenn Sie sie explizit mit `Select-Object` abfragen:

```
PS> Get-Process | Select-Object -Property Name, Company, Description, Path

Name     Company                    Description             Path
----     -------                    -----------             ----
AcroRd32 Adobe Systems Incorporated Adobe Acrobat Reader DC C:\Program…
chrome   Google LLC                 Google Chrome           C:\Program…
Code     Microsoft Corporation      Visual Studio Code      C:\Users\t…
(...)
```

Auf ähnliche Weise kann unsere Alternative zu `winver.exe` (siehe Seite 107) nun erheblich beschleunigt werden, denn auf Windows-Systemen lagern die meisten Informationen über das Betriebssystem in einer Datenbank, der *Windows-Registry*. Mit `Get-ItemProperty` kann man auf die darin enthaltenen Werte zugreifen. `Select-Object` wählt daraus die Registry-Werte aus, die Sie benötigen:

```
Get-ItemProperty -Path 'HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\' |
Select-Object -Property DisplayVersion, CurrentBuildNumber, UBR, RegisteredOwner,
RegisteredOrganization
```

Listing 2.11: Betriebssystem-Versionsinfos aus der Registry abrufen

Das Ergebnis könnte so oder ähnlich aussehen:

```
DisplayVersion         : 20H2
CurrentBuildNumber     : 19042
UBR                    : 685
RegisteredOwner        : Tobias
RegisteredOrganization : psconf.eu
```

Mehr Informationen zu `Get-ItemProperty` und der Windows-Registry erhalten Sie in Kapitel 7.

Unerwünschte Zeilen herausfiltern

`Where-Object` legt fest, welche Zeilen (oder Objekte) Sie benötigen und welche nicht. Geben Sie einfach das Kriterium an, das erwünschte Objekte erfüllen müssen.

`Get-Service` liefert zum Beispiel alle Dienste (bei Linux *Daemons* genannt). Die folgende Zeile filtert mit dem Vergleichsoperator `-eq` alle Dienste heraus, bei denen die Eigenschaft `Status` den Wert `Running` enthält:

```
PS> Get-Service | Where-Object Status -eq Running

Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Running  Appinfo            Anwendungsinformationen
```

```
Running  AppXSvc           AppX-Bereitstellungsdienst (AppXSVC)
(...)
```

Dabei steht `-eq` für das englische Wort *equals* (ist gleich).

Schauen wir uns an einem weiteren Beispiel an, wie nützlich Vergleichsoperatoren und `Where-Object` sind, um die Ergebnisse von Befehlen zu verfeinern. Das nächste Beispiel liefert nur Prozesse, die über ein eigenes Fenster verfügen (bei denen die Eigenschaft `MainWindowHandle` ungleich 0 ist). Von diesen werden dann der Name, die Prozess-ID und der aktuelle Text aus der Fenstertitelleiste angezeigt:

```
PS> Get-Process | Where-Object MainWindowHandle -ne 0 | Select-Object -Property Name, Id,
MainWindowTitle

Name              Id MainWindowTitle
----              -- ---------------
AcroRd32        5212 Windows PowerShell Language Specification Version …
Code            2096 netstatps7.ps1 - PowerShell - Visual Studio Code
explorer        4356 Windows
powershell     14900 Windows PowerShell
pwsh            3028 C:\myportablePowershell
Teams          12396 Online Training - Powershell Training Termin 1 | M…
```

In diesem Beispiel steht der Vergleichsoperator `-ne` für den englischen Begriff *not equal* (nicht gleich).

Beliebige Befehle verketten

Welche Befehle Sie mit dem Pipeline-Operator | verketten, ist im Grunde gleichgültig, es müssen nicht die speziellen PowerShell-SQL-Befehle von eben sein. Und Sie können auch mehr als nur zwei Befehle miteinander verketten.

Dialogfelder für den Helpdesk

Der jeweils auf das |-Zeichen folgende Befehl erhält dabei grundsätzlich das, was von seinen unmittelbaren Vorgängerbefehlen zurückgegeben wurde. Das Beispiel von eben könnte man also mit wenigen Handgriffen zu einem kleinen Helpdesk-Tool erweitern:

Die laufenden Programme werden von `Out-GridView` in einem Fenster zur Auswahl angeboten, und wenn Sie eine oder (bei festgehaltener Taste Strg) mehrere Anwendungen darin markieren und rechts unten auf *OK* klicken, werden nur die von Ihnen ausgewählten Prozesse an den nächsten Befehl weitergeleitet: `Stop-Process`. Dieser Befehl würde die ausgewählten Anwendungen also sofort und ohne weitere Nachfragen beenden:

```
Get-Process |
  Where-Object MainWindowHandle -gt 0 |
  Select-Object -Property Name, Id, MainWindowTitle |
  Out-GridView -Title 'Wählen Sie Prozesse aus, die beendet werden' -PassThru |
  Stop-Process -WhatIf
```

Listing 2.12: Anwendungsprogramme per Dialogfeld auswählen und beenden

Die Pipeline-Verkettung führt also oft zu sehr langen Zeilen. Damit sie lesbar bleiben, hat es sich eingebürgert, so wie in Listing 2.12 nach jedem |-Operator die Zeile zu umbrechen und die folgenden Zeilen ein wenig einzurücken. So wird deutlich, dass sie eine Fortsetzung der vorangegangenen Zeile sind.

Hinweis

Weil beim sofortigen Beenden von Prozessen mit `Stop-Process` neben anderen Dingen auch keine Zeit mehr bleibt, um schnell noch ungesicherte Daten zu speichern, droht hier Informationsverlust.

Deshalb simuliert `Stop-Process` im Beispiel nur und zeigt aufgrund des Parameters `-WhatIf` lediglich an, was es getan *hätte*. `-WhatIf` ist ein *allgemeiner Parameter* und steht bei jedem PowerShell-Befehl zur Verfügung, der etwas verändern und also Schaden anrichten könnte. Entfernen Sie im Code `-WhatIf`, wenn Sie es ernst meinen und wirklich Prozesse killen möchten.

Haben Sie übrigens die enge Verwandtschaft zwischen `Out-GridView` und `Where-Object` in den Beispielen davor bemerkt?

Wenn Sie bei `Out-GridView` den Parameter `-PassThru` angeben, wird sein Fenster zum Universal-Auswahldialog. `Where-Object` filtert also Objekte automatisch (basierend auf der mitgegebenen Bedingung), `Out-GridView` filtert sie manuell: Sie entscheiden per Klick, welche Objekte ausgewählt werden.

Und noch einmal vorsichtshalber der Hinweis: Das praktische `Out-GridView` steht nur unter Windows zur Verfügung, weil es Fenster verwendet. Auf Linux und macOS fehlt (derzeit noch) die Unterstützung für grafische Oberflächen.

Sonntagskinder: Wochentag eines Datums finden

Im nächsten Beispiel wird `Read-Host` mit `Get-Date` verbunden: `Read-Host` fragt nach Ihrem Geburtstag und gibt Ihre Eingabe als Ergebnis zurück. Diese Eingabe wird mit dem |-Operator weitergeleitet an `Get-Date`, das sie in ein Datum verwandelt und mithilfe des Parameters `-Format` einen formatierten Text mit dem berechneten Wochentag zurückgibt.

So können Sie feststellen, ob Sie ein Sonntagskind sind:

```
Read-Host -Prompt 'Geburtstag' | Get-Date -Format '"Sie sind ein" dddd"s-Kind!"'
```

Listing 2.13: Wochentag bestimmen, auf den ein Datum fällt

Das Ergebnis sieht dann so aus:

```
Geburtstag: 3.12.78
Sie sind ein Sonntags-Kind!
```

Beliebige Dateien in ein ZIP-Archiv verpacken

Und auch handfeste Limitationen von Cmdlets lassen sich über die Verkettung ganz einfach lösen: `Compress-Archive` kann eigentlich nur Dateien aus einem einzelnen Ordner in eine ZIP-Datei packen. Schlecht.

Allerdings kann es alle Dateien in ein ZIP-Archiv packen, die man dem Cmdlet über die Pipeline zusendet. Gut!

Möchten Sie also Dateien auch aus allen Unterordnern oder Dateien aus mehreren verschiedenen Ordnern in einer einzelnen ZIP-Datei verpacken, teilen Sie die Aufgabe auf und setzen lieber ein Team ein:

- Lassen Sie den Spezialisten `Get-ChildItem` zuerst die gewünschten Dateien heraussuchen, die ins ZIP-Archiv gepackt werden sollen. `Get-ChildItem` unterstützt rekursive Dateisuchen und auch mehrere Stammpfade.
- Leiten Sie diese Dateien an den Verpackungsspezialisten `Compress-Archive` weiter, um sie ins ZIP-Archiv zu verfrachten.

Hier ein Beispiel: Listing 2.14 sammelt sämtliche PowerShell-Skripte (Dateierweiterung *.ps1*), die sich irgendwo in Ihrem Benutzerprofil befinden (die vordefinierte Variable `$home` liefert den Pfad), und verpackt sie in einer ZIP-Datei. Falls Sie noch gar keine PowerShell-Skripte erstellt haben, ändern Sie die Dateierweiterung und »zippen« andere Dateitypen, zum Beispiel Bilder oder Word-Dokumente:

```
# Ordner anlegen, in dem die ZIP-Datei gespeichert werden soll:
New-Item -Path c:\meineZipDateien -ItemType Directory -ErrorAction Ignore

# Alle PowerShell-Skripte im Benutzerprofil suchen
# (ändern Sie den Dateityp und "zippen" Sie andere Dateiarten,
# falls Sie noch gar keine PowerShell-Skripte besitzen):
Get-ChildItem -Path $home -Filter *.ps1 -Recurse -ErrorAction Ignore |
# In eine ZIP-Datei speichern:
Compress-Archive -DestinationPath 'c:\meineZipDateien\meineSkripte.zip'

# Datei anzeigen
Get-Item -Path 'c:\meineZipDateien\meineSkripte.zip'
```

Listing 2.14: Alle PowerShell-Skripte aus dem eigenen Benutzerprofil in eine ZIP-Datei speichern

Klassische Variablen: »Türchen 2«

Auch mit Variablen kann man verschiedene Befehle verbinden, und tatsächlich ist vielen dieses klassische Prinzip aus anderen Programmiersprachen vertraut. Hierbei speichert man die Ergebnisse des ersten Befehls in einer Variablen und übergibt diese dann einem Parameter des zweiten Befehls.

Listing 2.13 hatte den Wochentag bestimmt, auf den ein bestimmtes Datum fiel, indem `Read-Host` und `Get-Date` über die Pipeline direkt miteinander verbunden wurden. Das sah so aus:

```
PS> Read-Host -Prompt 'Geburtstag' | Get-Date -Format '"Sie sind ein" dddd"s-Kind!"'
```

Dasselbe kann man auch mit Variablen erreichen, und diesmal transportieren Variablen die Informationen von einem Befehl zum nächsten:

```
$eingabe = Read-Host -Prompt 'Geburtstag'
Get-Date -Date $eingabe -Format '"Sie sind ein" dddd"s-Kind!"'
```

Listing 2.15: Wochentagsbestimmung mit Variablen anstelle der Pipeline

Listing 2.15 speichert das Resultat von `Read-Host` in der Variablen `$eingabe` und weist sie danach dem Parameter `-Date` des Befehls `Get-Date` zu.

Eine Variable beginnt in PowerShell stets mit `$`, gefolgt von einem beliebigen Namen, und funktioniert im Grunde wie eine Schublade, in der Sie beliebige Informationen aufbewahren können:

```
PS> $datum = Get-Date          # Türchen 2: alle Ergebnisse in einer Variablen speichern
PS> $datum | Out-GridView      # Türchen 1: strukturierte Daten, Zeit bleibt gleich
PS> $datum | Out-GridView      # Türchen 1: strukturierte Daten, Zeit bleibt gleich
PS> $datum                     # Türchen 3: reiner Text, lokalisiert
```

Ebenso wie bei der Pipeline speichert auch eine Variable die Ergebnisse eines Befehls verlustfrei und ohne Kürzungen oder Textumwandlungen. Sie können deshalb die Informationen, die in der Variablen gespeichert sind, ebenso verlustfrei zu jedem beliebigen Zeitpunkt an andere

Befehle weiterleiten. Erst wenn Sie sie über »Türchen 3« in die Konsole ausgeben, wird reduzierter Text daraus.

Weil die Variable die Zeitinformation in genau dem Originalformat speichert, das schon im GridView sichtbar geworden ist, dürfen Sie mit dem Punkt (.) auf diese Einzelinformationen zugreifen und könnten so gezielt beispielsweise den aktuellen Wochentag (unabhängig von Spracheinstellungen immer in Englisch) ermitteln:

```
PS> $datum.DayOfWeek
Thursday
```

Hier zeigt sich der enorme Vorteil einer »objektorientierten« Shell: Weil die Ergebnisse als strukturiertes Objekt vorliegen, können Sie gezielt auf die gewünschten Einzelinformationen zugreifen, zum Beispiel `DayOfWeek`.

Anders als bei textbasierten Shells brauchen Sie also keine Tools wie *grep* oder aufwendige Textbefehle mit sogenannten *regulären Ausdrücken* einzusetzen, um an die gewünschten Einzelinformationen zu gelangen.

Profitipp

PowerShell unterstützt also zwei Möglichkeiten, Befehle miteinander zu kombinieren: die Pipeline und Variablen. Ist es reine Geschmackssache, für welche Variante Sie sich entscheiden?

Nein, beides sind unterschiedliche Techniken mit spezifischen Vor- und Nachteilen. Bei der Pipeline wird jedes Ergebnis einzeln wie bei einer Feuerwehr-Eimerkette zuerst von allen gekoppelten Befehlen bearbeitet, bevor das jeweils nächste Ergebnis an die Reihe kommt. Die Pipeline benötigt also wesentlich weniger Speicherplatz als Variablen, weil stets nur ein Ergebnis im Speicher gehalten werden muss, und liefert erste Ergebnisse in Echtzeit.

Wirklich entscheidend ist das aber bloß, wenn ein Befehl sehr viele Ergebnisse liefert. Bei nur einigen Hundert Ergebnissen ist es also im Grunde doch Geschmackssache, welche Variante Ihnen lieber ist.

Neue Befehle nachrüsten

Sie haben es gesehen: Wenn für eine bestimmte Aufgabe die passenden Cmdlets bereitstehen, ist Automation nicht schwierig. Sie haben auch bereits gesehen, dass man selbst kompliziertesten Code als einfach zu verwendendes Cmdlet verpacken kann.

PowerShell kommt mit einem Grundstock allgemeiner Cmdlets, aber rund um den Globus arbeiten Menschen an echten Automationsproblemen und entwickeln dabei teils in monatelanger Arbeit hoch professionelle Lösungen. Dank des Legokonzepts der PowerShell entstehen dabei nicht monolithische Speziallösungen, sondern wiederverwertbare neue Universalbefehle.

Damit also nicht ständig das Rad neu erfunden werden muss, betreibt Microsoft mit der *PowerShell Gallery* (*https://www.powershellgallery.com*) einen kostenlosen Onlinemarktplatz, auf dem solche Praxislösungen von der PowerShell-Community getauscht werden können.

Wie sehr es sich also lohnt, die Grundlagen der PowerShell in diesem Kapitel kennenzulernen, sollen zum Abschluss einige bunt gemischte Praxisbeispiele beweisen:

Mit dem bis zu dieser Stelle gesammelten Wissen können Sie bereits die teils extrem komplexen und leistungsfähigen Automationslösungen anderer IT-Profis für eigene Zwecke nutzen

und so ganz erheblich Zeit sparen. Kompliziert ist das dank des erweiterbaren Cmdlet-Konzepts der PowerShell überhaupt nicht.

Beispiel 1: QR-Codes generieren

Falls Sie auf Ihre nächste Geburtstagseinladung einen QR-Code drucken wollen, mit dem sich die Gäste eine Wegbeschreibung anzeigen lassen können, und falls Sie Ihren Gästen dann auf der Party per Scan eines weiteren QR-Codes bequem Zugang zu Ihrem WLAN gewähren möchten, brauchen Sie rein gar keine Spezialkenntnisse dazu, wie QR-Codes generiert werden. Sie brauchen lediglich die passenden Cmdlets und das allgemeine PowerShell-Wissen aus diesem Kapitel.

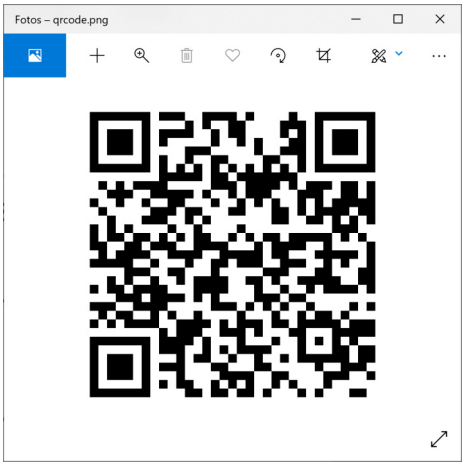

Abbildung 2.18: Neu hinzugefügte Cmdlets erzeugen QR-Codes für Smartphones.

So rüsten Sie die nötigen Cmdlets von der offiziellen *Microsoft PowerShell Gallery* nach:

```
PS> Install-Module -Name QRCodeGenerator -Scope CurrentUser -Force ↵
```

Wichtig

Achten Sie, wie in Kapitel 1 beschrieben, darauf, die Skriptausführung einzuschalten, andernfalls funktionieren viele Module nicht. Geben Sie im Zweifelsfall ein:

```
PS> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned ↵
```

Sollte `Install-Module` bei Ihnen nicht funktionieren, erfahren Sie ebenfalls in Kapitel 1, ob alle dafür notwendigen Komponenten auf dem neuesten Stand sind. Bei älteren Windows-Versionen muss gegebenenfalls das neueste Sicherheitsprotokoll TLS 1.2 aktiviert werden.

Wenn Sie diesen Befehl zum ersten Mal ausführen, wird PowerShell Ihr Einverständnis einholen, um die notwendige Erweiterung aus dem Internet herunterzuladen. Danach wird das PowerShell-Modul *QRCodeGenerator* heruntergeladen.

Wichtig

Microsoft überwacht die *PowerShell Gallery* natürlich und prüft den Code, der dort hochgeladen und bereitgestellt wird. Garantien oder Support gibt es für dieses kostenlose Angebot allerdings nicht: Wer von dort etwas herunterlädt, ist selbst dafür verantwortlich.

Wenige Sekunden später haben Sie neue PowerShell-Befehle, zum Beispiel diesen hier:

```
PS> New-QRCodeWifiAccess -SSID myhotspot -Password TOPSECRET12 -Show ↵
```

Passen Sie den Namen und das Zugangskennwort Ihres WLAN an. Dann führen Sie den Befehl mit ↵ aus.

Es öffnet sich im Webbrowser ein QR-Code, und wenn Sie diesen Code mit der Kamera des Smartphones anvisieren, erhalten Sie die Möglichkeit, die hinterlegten Zugangsdaten für das Wi-Fi-Netz zu speichern.

Oder Sie schauen sich die Parameter des Cmdlets an. Wenn Sie mögen, können Sie QR-Code-Grafiken nämlich auch unbeaufsichtigt als PNG-Dateien abspeichern:

```
PS> New-QRCodeWifiAccess -SSID myhotspot -Password TOPSECRET12 -OutPath $home\code.png
```

Mit `Get-Command` können Sie sich auch die übrigen Cmdlets für andere QR-Code-Typen auflisten lassen:

```
PS> Get-Command -Module QRCodeGenerator

CommandType Name                       Version Source
----------- ----                       ------- ------
Alias       New-QRCodeGeolocation      2.4.0   QRCodeGenerator
Alias       New-QRCodeText             2.4.0   QRCodeGenerator
Alias       New-QRCodeTwitter          2.4.0   QRCodeGenerator
Alias       New-QRCodeVCard            2.4.0   QRCodeGenerator
Alias       New-QRCodeWifiAccess       2.4.0   QRCodeGenerator
Function    New-PSOneQRCodeGeolocation 2.4.0   QRCodeGenerator
Function    New-PSOneQRCodeText        2.4.0   QRCodeGenerator
Function    New-PSOneQRCodeTwitter     2.4.0   QRCodeGenerator
Function    New-PSOneQRCodeVCard       2.4.0   QRCodeGenerator
Function    New-PSOneQRCodeWifiAccess  2.4.0   QRCodeGenerator
```

Hinweis

Wenn Sie mit `New-QRCodeGeolocation` einen QR-Code für eine Örtlichkeit anlegen lassen wollen, erwartet das Cmdlet entweder eine Adresse oder eine Längen- und Breitengradangabe von Ihnen, beispielsweise:

```
PS> New-QRCodeGeolocation -Latitude 52.34897 -Longitude 9.81441 -Show
```

Beispiel 2: Automatische HTML-Reports

Wenn Sie Befehlsergebnisse als professionellen HTML-Report an den Chef oder Kollegen weiterreichen wollen, brauchen Sie dank PowerShell und einer entsprechenden Befehlserweiterung überhaupt kein Spezialwissen zu HTML oder JavaScript. Sie brauchen sich nur um die Daten zu kümmern, die Sie im HTML-Report anzeigen wollen.

Mit den passenden Cmdlets generieren Sie dann anspruchsvolle HTML-Reports wie den in Abbildung 2.19, einschließlich PDF-, Excel- und CSV-Export über die Schaltflächen in der oberen linken Ecke, und können sich auf das konzentrieren, was Sie wirklich interessiert: den Inhalt des Reports.

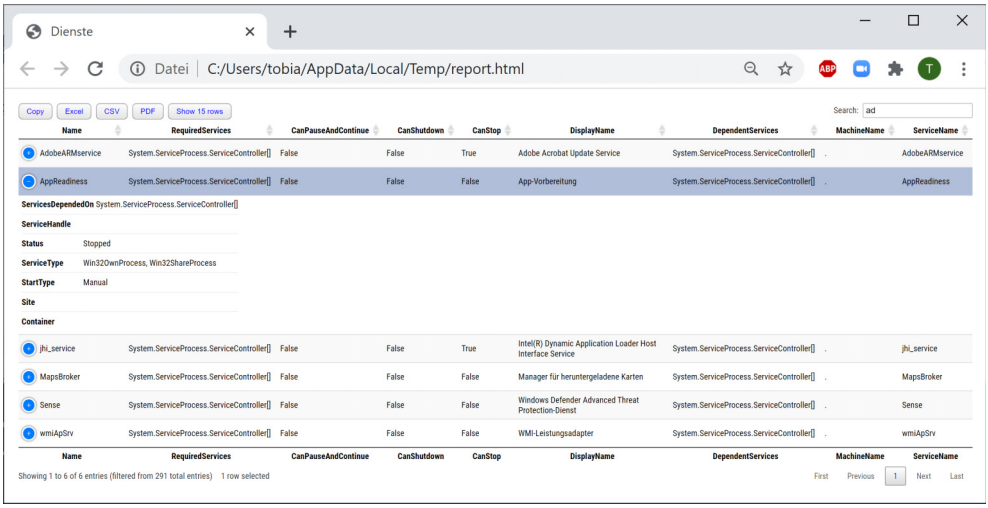

Abbildung 2.19: Einen schicken und durchsuchbaren HTML-Report mit PDF-Export in wenigen Sekunden

Eine Tabelle als HTML-Report

Das Modul, das die notwendigen Cmdlets nachrüstet, heißt *PSWriteHTML* und wird wie eben mit `Install-Module` von der *PowerShell Gallery* heruntergeladen und installiert:

```
PS> Install-Module -Name PSWriteHTML -Scope CurrentUser -Force
```

Dieses Modul liefert eine ganze Reihe neuer Cmdlets, aber für den Anfang brauchen Sie nur zwei davon. Führen Sie diese Befehle aus, um einen ersten Testreport zu generieren:

```
$path = "$env:temp\report.html"
$daten = Get-Service
$tabelle = { New-HTMLTable -DataTable $daten }
New-HTML -HtmlData $tabelle -FilePath $path -ShowHTML -TitleText 'Dienste'
```

Listing 2.16: Einen HTML-Report erstellen

Abbildung 2.16 generiert einen HTML-Report mit allen Diensten Ihres Computers (siehe Abbildung 2.19). Der Report öffnet sich nach ein paar Sekunden in Ihrem Browser:

- Oben rechts können Sie die Daten in Echtzeit filtern.
- Oben links finden Sie fix und fertige Exportmöglichkeiten und könnten den Report sofort als PDF-Datei weitergeben.
- Links vor jeder Zeile können Sie sich per Klick die Details des Datensatzes anzeigen lassen.
- In der Fußzeile findet sich die Paginierung, mit der seitenweise vor- und zurückgeblättert wird.

Mehrere Tabellen anzeigen

Wenn Sie sich Abbildung 2.16 genauer ansehen, fällt neben vielem bereits Bekanntem auch etwas Neues auf, was kein sonderlich großes Wunder ist – immerhin befinden Sie sich erst im zweiten Kapitel, und es folgen noch viele weitere.

Neu sind die geschweifte Klammern. Während runde Klammern Code zusammenfassen, der *sofort* ausgeführt wird, fassen *geschweifte* Klammern Code zusammen, der *nicht sofort* ausgeführt wird – sondern beispielsweise an jemand anderen weitergegeben werden kann, der ihn dann unter seiner Regie ausführt.

Die Variable `$tabelle` enthält also den Code, der den Inhalt der HTML-Seite gestaltet, und dieser Code wird an `New-Html` weitergereicht. `New-Html` führt den von Ihnen übergebenen Code dann aus, um die notwendigen Daten für den Report zu generieren, und erzeugt die HTML-Seite.

Mit diesem Wissen und dem Rest der Erkenntnisse aus diesem Kapitel könnten Sie auch sehr viel anspruchsvollere Reports gestalten, zum Beispiel solche mit mehreren Tabellen:

```
$path = "$env:temp\report.html"

# Dieser Code wird an New-Html weitergereicht, der damit den
# Inhalt der HTML-Seite gestaltet:
$tabelle = {

  # Alle Dienste abrufen und einige Spalten auswählen:
  $daten = Get-Service | Select-Object -Property DisplayName, Status, StartType
  # Tabelle erzeugen:
  New-HTMLTable -DataTable $daten

  # Alle Prozesse auflisten, die ein Fenster haben:
  $daten = Get-Process |
    Where-Object MainWindowTitle |
    Select-Object -Property Name, Company, Description, MainWindowTitle, CPU
  # Tabelle erzeugen:
  New-HTMLTable -DataTable $daten

  }

# HTML-Seite generieren und anzeigen:
New-HTML -HtmlData $tabelle -FilePath $path -ShowHTML -TitleText 'Dies und das' -Online
```

Listing 2.17: Einen HTML-Report mit zwei Tabellen generieren

Und auch die weiteren Parameter von `New-Html` sind nützlich: Gibt man wie in Listing 2.17 `-Online` an, werden die JavaScript-Bibliotheken nicht in die HTML-Datei eingebunden, sondern online aus dem Web geladen. Das verkleinert die HTML-Datei erheblich, ist aber natürlich nur dann sinnvoll, wenn der Empfänger des HTML-Reports Internetzugang hat.

Konditionale Formatierung

Sie wissen inzwischen bereits, dass alle Cmdlets aus Modulen stammen, und selbst wenn Sie sich nicht mehr daran erinnern, dass die neuen Cmdlets wie `New-Html` aus dem nachgerüsteten Modul *PSWriteHtml* stammen, würde `Get-Command` es Ihnen jederzeit verraten:

```
PS> Get-Command -Name New-Html

CommandType Name     Version Source
----------- ----     ------- ------
Function    New-HTML 0.0.120 PSWriteHTML
```

Sie wissen mittlerweile auch, dass `Get-Command` die Cmdlets auflisten kann, die in einem Modul enthalten sind. So könnten Sie sich leicht darüber informieren, welche weiteren Cmdlets zur Gestaltung Ihrer HTML-Reports zur Verfügung stehen:

```
PS> Get-Command -Module PSWriteHtml | select -First 20

CommandType Name                           Version Source
----------- ----                           ------- ------
Alias       Add-CSS                        0.0.120 PSWriteHTML
Alias       Add-JavaScript                 0.0.120 PSWriteHTML
Alias       Add-JS                         0.0.120 PSWriteHTML
Alias       New-ChartCategory              0.0.120 PSWriteHTML
Alias       New-Diagram                    0.0.120 PSWriteHTML
Alias       New-DiagramEdge                0.0.120 PSWriteHTML
Alias       New-DiagramOptionsEdges        0.0.120 PSWriteHTML
Alias       New-HTMLColumn                 0.0.120 PSWriteHTML
Alias       New-HTMLContent                0.0.120 PSWriteHTML
Alias       New-HTMLLink                   0.0.120 PSWriteHTML
Alias       New-HTMLPanelOption            0.0.120 PSWriteHTML
(...)
```

Es sind so viele, dass man sich mit ihnen getrost ein paar Tage beschäftigen könnte, aber schnell werden Sie entdecken, dass man damit kinderleicht hoch anspruchsvolle Reports erstellen kann.

Mit `New-TableCondition` lassen sich zum Beispiel Schrift- und Hintergrundfarbe von Tabellenzellen und -reihen automatisch anpassen, basierend auf Text- oder Zahlenvergleichen.

Listing 2.18 generiert zum Beispiel einen Report mit allen Diensten. Nicht laufende Dienste (deren *Status* den Inhalt *Stopped* hat) werden mit rotem Hintergrund gekennzeichnet. Und ist ein Dienst abgeschaltet (*StartType* steht auf *Disabled*), wird der Starttyp in orangefarbener Schrift formatiert (siehe Abbildung 2.20):

```
$path = "$env:temp\report.html"

$inhalt = {
  # Alle Dienste abrufen ...
  $services = Get-Service |
    # und nützliche Eigenschaften anzeigen:
    Select-Object -Property Name, DisplayName, Status, StartType

  # Eine neue Tabelle mit diesen Diensten generieren (100 pro Seite):
  New-HTMLTable -DataTable $services -HideFooter -PagingLength 100 -Html {

    # Einige Zellen der Tabelle automatisch abhängig vom Zelleninhalt farbig hervorheben

    # und alle Zeilen, in denen in der Spalte "Status" der Text "Stopped" steht,
```

```
    # weiß auf rotem Hintergrund hervorheben:
    New-TableCondition -Name 'Status' -ComparisonType string -Operator eq -Value 'Stopped'
    -Color White -BackgroundColor Crimson -Row
    # Alle Zeilen, in denen in der Spalte "StartType" der Text "Disabled" steht,
    # in orangefarbener Schrift hervorheben:
    New-TableCondition -Name 'StartType' -ComparisonType string -Operator eq -Value 'Disabled'
    -Color Orange

  }
}

# Tabelle generieren und anzeigen:
New-HTML -Name 'Dienste-Übersicht' -FilePath $path -Show -HtmlData $inhalt
```

Listing 2.18: HTML-Report mit farbig hervorgehobenen Diensten, die nicht laufen oder abgeschaltet sind

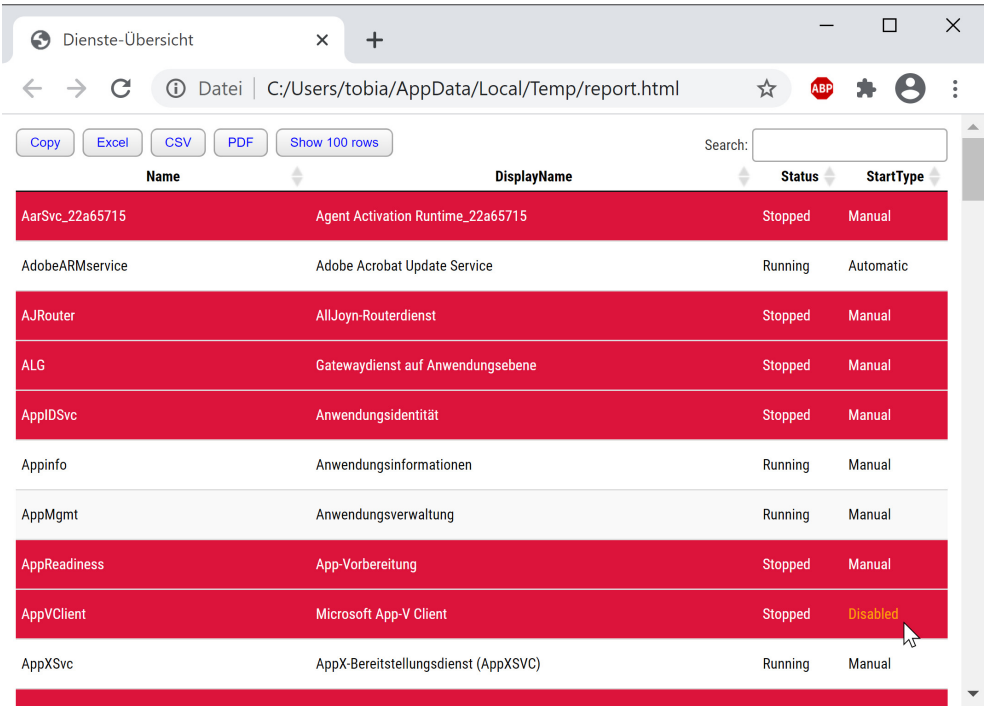

Abbildung 2.20: Farbige HTML-Reports mit hervorgehobenen gestoppten Diensten

Diagramme

Wie mächtig diese Cmdlet-Sammlung ist und mit wie wenig Code man damit sogar Diagramme und ganze Systemreports herstellen kann, soll ein letztes Beispiel demonstrieren:

```
$path = "$env:temp\report.html"

# Prozesse mit eigenem Fenster auswählen:
$Processes = Get-Process |
  Where-Object MainWindowTitle |
  Select-Object -Property ProcessName, Id, Company, Description, MainWindowTitle
```

```
New-HTML -TitleText 'Prozess-Diagramm' -Online -FilePath $path -ShowHTML -HtmlData {
    New-HTMLSection -Invisible -Content {
        # Seite in zwei Spalten unterteilen

        # Erste Spalte: die Tabelle mit den Prozessen:
        New-HTMLPanel -Content {
            New-HTMLTable -DataTable $Processes -DataTableID 'Table1'
        }
        # Zweite Spalte: ein Diagramm mit den Abhängigkeiten:
        New-HTMLPanel -Content {
            New-HTMLDiagram -Height '1000px' {
                $label = "Prozesse auf`r`n$env:computername"
                New-DiagramEvent -ID 'Table1' -ColumnID 1
                # In die Mitte ein Windows-Icon in "AirForceBlue" legen:
                New-DiagramNode -Label $label -IconBrands windows -IconColor AirForceBlue

                # Alle Prozesse mit Pfeilen mit diesem Symbol verbinden:
                $Processes | Foreach-Object {
                    $childLabel = $_.ProcessName + " (Pid " + $_.Id + ")"
                    New-DiagramNode -Label $childLabel -Id $_.Id -To $label
        -ArrowsFromEnabled -Shape box -LinkColor Apricot
                }
            }
        }
    }
}
```

Listing 2.19: Zweispaltiger HTML-Report mit einem Prozessdiagramm

Listing 2.19 erstellt einen eindrucksvollen zweispaltigen Prozessreport und zeigt die laufenden Prozesse als Grafik an (siehe Abbildung 2.21) – es kostet nur wenig Fantasie, sich vorzustellen, wie man auf diese Weise ganze Netzwerkinfrastrukturen dynamisch visualisieren könnte.

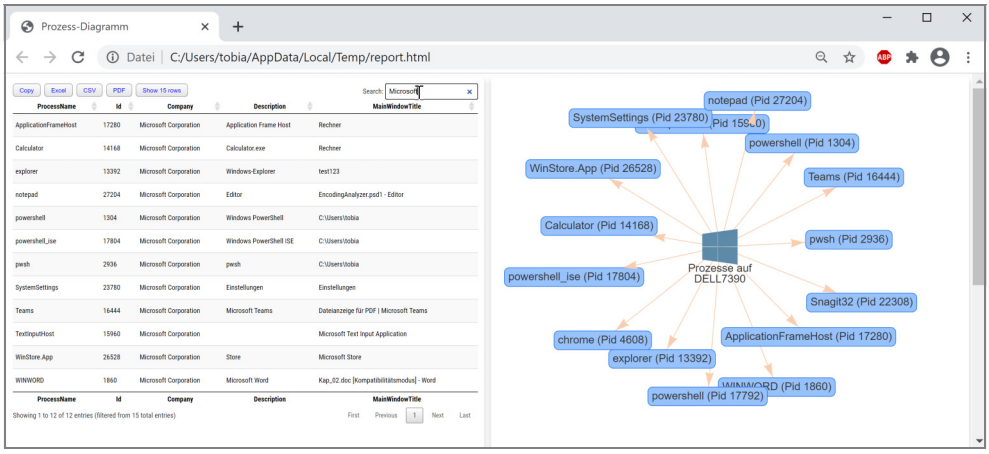

Abbildung 2.21: Eindrucksvolle Diagramme mit wenigen Zeilen PowerShell-Code

Tipp

Viele weitere teils sehr anspruchsvolle Reports und Anwendungsbeispiele hat der Autor der Cmdlets auf seiner Projektseite hier aufgelistet: *https://github.com/EvotecIT/PSWriteHTML#articles-worth-reading-to-understand-use-cases.*

Dort finden Sie auch den gesamten Quellcode. Der allerdings hat es in sich, und gerade weil man sich im Alltag eben nicht mit diesen Details befassen möchte, ist es so praktisch, dass all diese Komplexität durch Cmdlets von Ihnen abgeschirmt werden kann.

Beispiel 3: Musik spielen

PowerShell ist nicht etwa nur für Systemadministratoren spannend – man kann damit alles Mögliche steuern und automatisieren, natürlich auch im künstlerischen und privaten Umfeld. Schauen wir uns an, wie Musiker von PowerShell profitieren (und kreative Systemadministratoren künftig langwierige Skripte mit dem »Star Wars Imperial March« hinterlegen könnten).

Diesmal werden keine Cmdlets nachgerüstet, sondern Applications, und am Ende werden Sie nicht nur Musik abspielen können, sondern auch Partituren erzeugen und MIDI-Dateien in PDF-Notenblätter umwandeln können.

Beginnen wir aber simpel und lassen wir am Ende PowerShell ganze Partituren spielen – alles mit den Mitteln, die Sie inzwischen kennengelernt haben.

Einfache Töne spielen

Listing 2.20 spielt den »Star Wars Imperial March« über den Systemlautsprecher und zeigt nebenbei, wie ein PowerShell-Skript Rohdaten in einem beliebigen Format auswerten könnte.

```
$music = '440-500,440-500,440-500,349-350,523-150,440-500,349-350,523-150,440-1000,659-500,659-
500,659-500,698-350,523-150,415-500,349-350,523-150,440-1000'

$music.Split(',') |
  ForEach-Object {
    $frequency, $duration = $_.Split('-')
    [Console]::Beep($frequency, $duration)
    Write-Host $_
}
```

Listing 2.20: Star Wars Imperial March spielen

Die Variable `$music` enthält die zu spielenden Noten in Form eines langen Texts jeweils als Frequenz und Dauer (in Millisekunden). Die einzelnen Noten sind darin kommasepariert.

Dies könnte im Prinzip ebenso gut eine Textzeile aus einem Systemlogbuch sein, aus der Sie beispielsweise Servernamen extrahieren wollen: Es handelt sich also um Rohtext in einem bestimmten Format, das von PowerShell ausgewertet werden soll.

Die in jedem Textobjekt enthaltene Methode `Split()` kann den Text an einem beliebigen Zeichen aufsplitten und so durch das Splitten am Komma den langen Text aufteilen in eine Liste der einzelnen Noten.

Der Pipeline-Operator (|) leitet diese einzelnen Noten dann der Reihe nach an `Foreach-Object` weiter. Dieses Cmdlet führt für jedes eintreffende Element (also jede eintreffende Note) den

Code in den geschweiften Klammern aus. Das jeweils eintreffende Element selbst steht innerhalb der geschweiften Klammern stets in der speziellen Laufvariablen `$_` zur Verfügung. Darin ist beim ersten Durchlauf also `440-500` enthalten.

Weil `$_` wiederum Texte enthält (die einzelnen Bruchstücke, die `Split()` erzeugt hat), werden diese noch einmal gesplittet, diesmal am Minuszeichen »-«. Das Ergebnis sind zwei Informationen: die zu spielende Frequenz und die Dauer. Diese beiden werden in zwei separaten Variablen gespeichert: `$frequency` und `$duration`.

Die Methode `[Console]::Beep()`, die Sie ja schon kennen, spielt den Ton dann ab, und das Cmdlet `Write-Host` gibt die jeweils gespielte Note zur Kontrolle als Text in die Konsole aus.

Echte Noten angeben

Nun ist es etwas aufwendig, Noten in Frequenzen anzugeben. Deshalb unterstützt PowerShell Übersetzungstabellen, sogenannte *Hashtables*: In Listing 2.21 definiert `$scale` die Notennamen und weist ihnen jeweils ihre Frequenz zu. Mehr zu solchen Hashtables erfahren Sie in Kapitel 6.

```
# Noten in Frequenzen übersetzen:
$scale = @{
    D4b=297
    D4=293
    E4=329
    F4=349
    A4b=415
    A4=440
    B4b=466
    B4=493
    C4=523
    D5=587
    D5b=554
    E5b=622
    E5=659
    F5=698
    G5b=739
    G5=783
    A5b=830
    A5=880
}

$imperialmarch = 'A4-4,A4-4,A4-4,F4-8,C4-8,A4-4,F4-8,C4-8,A4-2,E5-4,E5-4,E5-4,F5-8,C4-8,A4b-4,
F4-8,C4-8,A4-2,A5-4,A4-8,A4-8,A5-4,A5b-8,G5-8,G5b-8,F5-8,G5b-8,B4b-8,E5b-4,D5-8,D5b-8,C4-8,
B4-8,C4-8,F4-8,A4b-4,F4-8,A4-8,C4-4,A4-8,C4-8,E5-2,A5-4,A4-8,A4-8,A5-4,A5b-8,G5-8,G5b-8,F5-8,
G5b-8,B4b-8,E5b-4,D5-8,D5b-8,C4-8,B4-8,C4-8,F4-8,A4b-4,F4-8,C4-8,A4-4,F4-8,C4-8,A4-4,F4-8,C4-8,A4-2'

# Länge einer ganzen Note in Millisekunden:
$fullLength = 2000

$imperialmarch.Split(',') |
  ForEach-Object {
    $note, $duration = $_.Split('-')
    [Console]::Beep($scale[$note], ($fullLength/$duration))
    Write-Host $_
    # Einen kurzen Schlafbefehl einsetzen, damit die Tonausgabe synchron bleibt
    # (technisches Detail des Beep()-Befehls, der leicht überfordert werden kann)
    Start-Sleep -MilliSeconds 200
  }
```

Listing 2.21: Echte Noten in Töne umwandeln

Nun können die Noten des »Imperial March« auch ohne Physikstudium angegeben werden. Anstelle der Frequenz darf jetzt die Musiknote verwendet werden, und anstelle von festen Zeiten in Millisekunden wird als zweite Information die Notenlänge relativ angegeben (`4` steht für eine Viertelnote).

Jetzt kann die gesamte Spielgeschwindigkeit zentral über `$fullLength` gesteuert werden, indem darin die gewünschte Dauer einer ganzen Note vermerkt wird. Sie können das Stück nun also schneller und auch langsamer abspielen lassen.

Der übrige Teil des Skripts ist fast unverändert geblieben. Diesmal allerdings werden die einzelnen Noten noch »übersetzt«: `$scale['Notenname']` ermittelt die Frequenz für einen Notennamen, und `($fullLength/'Notenlänge')` berechnet die Dauer einer Note in Millisekunden.

Imperial March als Orchesterpartitur spielen

`[Console]::Beep()` sorgt nicht gerade für einen Ohrenschmaus. Wenn Sie mehr möchten, als einfache Pieptöne abzuspielen, dann verwenden Sie den Synthesizer in Ihrer Soundkarte und füttern ihn mit der Partitur des »Imperial March«.

Solche Partituren finden Sie im Internet zuhauf, wenn Sie nach den Schlüsselwörtern »MIDI« und dem Namen des gesuchten Stücks googeln, denn »MIDI« unterstützt digitale Notenblätter.

```
# URL der MIDI-Datei für den "Imperial March"
# Achtung: Internetadressen veralten schnell. Wenn dieser Download nicht mehr
# zur Verfügung steht, googeln Sie wie angegeben selbst und verwenden die
# Internetadresse einer anderen MIDI-Datei.
$url = 'http://www.midis101.com/midi_download/8424/7629E74E55697C4C30B408F80B9077EF/
Star_Wars___Imperial_March'
$filename = Join-Path -Path $env:temp -ChildPath 'imperialmarch.mid'

# Datei herunterladen:
Invoke-RestMethod -UseBasicParsing -Uri $url -OutFile $filename

# Heruntergeladene Datei anzeigen:
Get-Item -Path $filename
```

Listing 2.22: Digitales Notenblatt für den Imperial March herunterladen

Sobald Sie das digitale Notenblatt als Datei mit der Dateierweiterung *.mid* heruntergeladen haben, kann PowerShell es an den Synthesizer Ihrer Soundkarte senden:

```
# Pfadname zur vorher heruntergeladenen MIDI-Datei:
$filename = Join-Path -Path $env:temp -ChildPath 'imperialmarch.mid'

# MIDI-Befehle nachladen:
Add-Type -AssemblyName PresentationCore

# Neuen Mediaplayer beschaffen:
$player = [System.Windows.Media.MediaPlayer]::new()

# MIDI-Datei öffnen, abspielen und schließen:
$player.Open($filename)

# MIDI-Datei wird asynchron im Hintergrund abgespielt,
# PowerShell läuft weiter. Sie könnten also MIDI-Dateien
# im Hintergrund abspielen, während Ihr PowerShell-Skript
# andere Aufgaben bearbeitet.
$player.Play()

# In diesem Fall hat PowerShell nichts weiter zu tun und
```

```
# wartet also, bis der Anwender mit Enter genug gehört hat:
$null = Read-Host -Prompt 'Drücken Sie ENTER, um zu beenden'

$player.Stop()

# Wichtig: Nur wenn der MIDI-Synthesizer geschlossen wird, können andere
# Programme wieder MIDI-Daten abspielen. Solange MIDI von einem
# Programm genutzt wird, ist die MIDI-Ausgabe anderer Programme
# nicht zu hören:
$player.Close()
```

Listing 2.23: Eine MIDI-Musikdatei mit PowerShell abspielen

Tipp

MIDI ist eine wichtige Technik für Profimusiker: *.mid*-Dateien enthalten digitale Notenblätter, die dann digital an Instrumente weitergeleitet werden – wenn Sie keine besitzen, spielt notfalls der MIDI-Synthesizer Ihrer Soundkarte.

Umgekehrt geht es auch: Wer ein Keyboard besitzt, kann die darauf gespielte Musik via MIDI direkt in Notenblättern mitschreiben lassen.

Notenblätter anzeigen und ausdrucken

Möchten Sie die tatsächliche Partitur in der heruntergeladenen *.mid*-Datei ansehen (oder vielleicht sogar verändern oder zum Mitsummen ausdrucken), installieren Sie sich einen MIDI-Editor, zum Beispiel das hervorragende und kostenlose *MuseScore* (Download unter *https://musescore.org/de*).

Tipp

Falls Sie in Kapitel 1 die Paketverwaltung *Scoop* installiert haben, können Sie *MuseScore* sogar vollautomatisch über die PowerShell herunterladen und installieren:

```
PS> scoop install musescore
Installing 'musescore' (3.6.0.487915773) [64bit]
MuseScore-3.6.0.487915773-x86_64.msi (106,5 MB)
[============================================================] 100%
Checking hash of MuseScore-3.6.0.487915773-x86_64.msi ... ok.
Extracting MuseScore-3.6.0.487915773-x86_64.msi ... done.
Running pre-install script...
Linking ~\scoop\apps\musescore\current => ~\scoop\apps\musescore\3.6.0.487915773
Creating shim for 'MuseScore'.
Creating shim for 'mscore'.
Creating shortcut for MuseScore (MuseScore.exe)
'musescore' (3.6.0.487915773) was installed successfully!
```

Wie immer bei *Scoop* wird die Anwendung als portable App im zentralen Ordner von *Scoop* gelagert. Starten Sie die Anwendung entweder manuell aus dem entsprechenden Ordner (und heften Sie sie danach an die Taskleiste) oder legen Sie eine Verknüpfung an eine gut erreichbare Stelle:

```
Explorer /select,$home\Scoop\Apps\musescore\current\bin\musescore.exe
```

Oder starten Sie die Anwendung direkt aus PowerShell, zum Beispiel mit dem Cmdlet `Start-Process`:

```
Start-Process -FilePath $home\Scoop\Apps\musescore\current\bin\musescore.exe
```

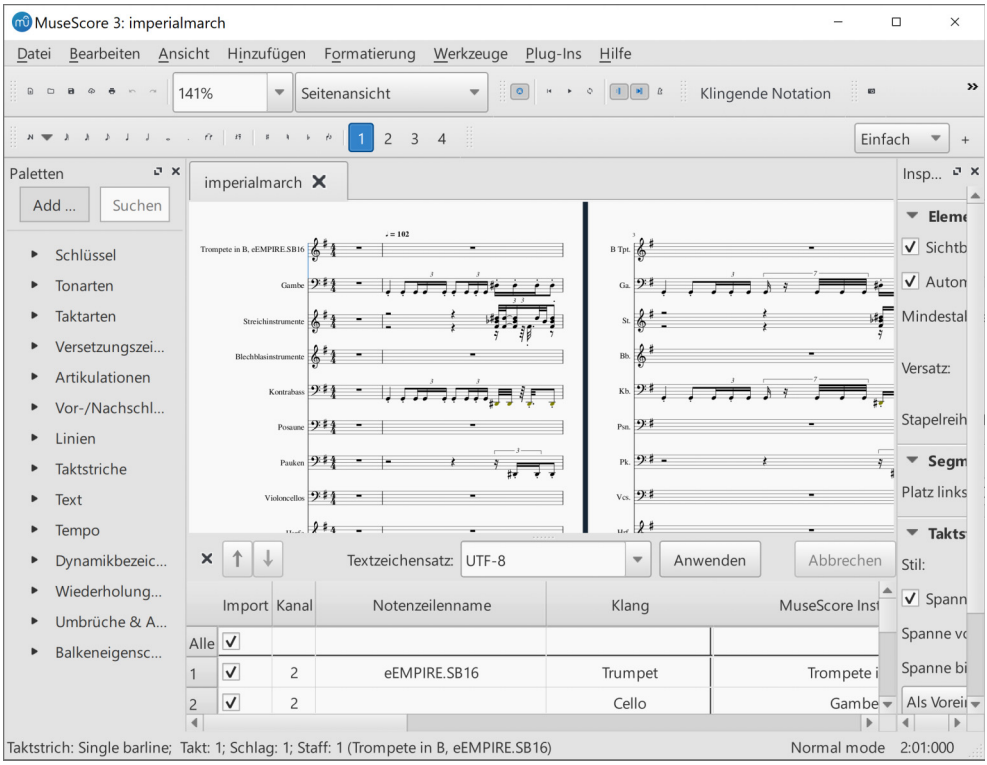

Abbildung 2.22: MuseScore kann MIDI-Dateien anzeigen und bearbeiten und ist automatisierbar.

Mit diesem Programm lassen sich *.mid*-Dateien öffnen, anzeigen und sogar bearbeiten. Laden Sie einfach die von Abbildung 2.22 heruntergeladene *.mid*-Datei in *MuseScore*.

Wollen Sie bearbeitete Partituren wieder als *.mid*-Datei speichern, damit PowerShell sie mit Ihren künstlerischen Änderungen erneut abspielen kann, wählen Sie im Programm *Datei/Export* und exportieren die Partition als *Standard MIDI-Datei*.

»MuseScore« als Automationsbefehl einsetzen

MuseScore ist ein weiteres Beispiel für eine Application, die den Befehlswortschatz der PowerShell bereichert, denn dieses Programm öffnet nicht nur ein grafisches Fenster für Endanwender. Sie können diese Anwendung auch mit Parametern automatisiert steuern. Sobald Sie also Anwendungen nachinstallieren, erweitern Sie damit gleichzeitig auch den Befehlswortschatz der PowerShell.

Dazu müssen Sie allerdings zuerst herausfinden, wo genau die Anwendung installiert wurde und wie der Programmname heißt. Falls Sie *MuseScore* über *Scoop* installiert haben, ist alles klar und (einigermaßen) einheitlich, denn eben haben Sie bereits den zentralen Aufbewahrungsort aller über *Scoop* installierten Programme kennengelernt. Der Pfadname lautet dann:

```
$home\Scoop\Apps\musescore\current\bin\musescore.exe
```

In allen anderen Fällen behelfen Sie sich mit einem PowerShell-Trick und starten zunächst einfach die Anwendung auf ganz beliebige Weise.

Sobald die Anwendung läuft, verwenden Sie `Get-Process`, um alle laufenden Prozesse aufzulisten, und fischen den Prozess heraus, der »Muse« im Namen führt (siehe Listing 2.24). Von ihm lassen Sie sich dann die Eigenschaft `Path` ausgeben und wissen nun, wie die Application exakt heißt und wo sie sich befindet:

```
# Achtung: MuseScore muss bereits laufen!
Get-Process *muse* | Select-Object -Property Name, Path
```

Listing 2.24: Speicherort einer laufenden Anwendung ermitteln

Auch dieses Beispiel verdeutlicht, dass die Arbeit mit Applications zwar viel Nutzen bringen kann, aber wegen der fehlenden Konsistenzen nicht immer nervenschonend ist. Wenn Sie beispielsweise *MuseScore* nicht über *Scoop* installiert haben, sondern ein Installationsprogramm aus dem Internet verwendeten, können sowohl der Speicherort als auch der Name der Application anders lauten.

Die Application heißt dann zum Beispiel nicht `musescore`, sondern `musescore3`, und befindet sich bei klassischer Installation üblicherweise in Windows-Systemen im Ordner *c:\program files*.

```
Name       Path
----       ----
MuseScore3 C:\Program Files\MuseScore 3\bin\MuseScore3.exe
```

Notenblätter als PDF-Dateien herstellen

Sobald Sie wissen, wie die Application heißt und wo sie sich befindet, können Sie damit beginnen, Aufgaben zu automatisieren. Dazu soll eine aus dem Internet heruntergeladene *.mid*-Datei als vollwertige Partition in Form einer PDF-Datei gespeichert werden. Sie erhalten so also aus kostenlosen und zuhauf verfügbaren *.mid*-Dateien wertvolle Notenblätter, und wenn Sie Heimmusikant sind, könnten Sie nun jeden Abend ein neues Stück proben, ohne neue Noten einkaufen zu müssen.

```
# Speicherort für heruntergeladene MIDI-Datei:
$MidiFile = Join-Path -Path $env:temp -ChildPath 'imperialmarch.mid'

# Dateiname der PDF-Datei, die generiert werden soll:
$PdfFile = Join-Path -Path $env:temp -ChildPath 'imperialmarch.pdf'

# Notenblätter als MIDI-Datei herunterladen:
Invoke-RestMethod -UseBasicParsing -Uri $url -OutFile $MidiFile

# Pfadname von MuseScore (muss installiert sein)
# ACHTUNG: PASSEN SIE DIESEN PFAD WIE IM KAPITEL BESCHRIEBEN AN!
# ER MUSS AUF IHRE KONKRETE MUSESCORE-ANWEISUNG WEISEN!
$muse = 'C:\Program Files\MuseScore 3\bin\MuseScore3.exe'
$exists = Test-Path -Path $muse
if ($exists -eq $false)
{
  # Wenn Sie MuseScore über scoop.exe installiert haben, ist dieser Pfad richtig:
  $muse = "$home\Scoop\Apps\musescore\current\bin\musescore.exe"
  $exists = Test-Path -Path $muse
  if ($exists -eq $false)
  {
    throw 'MuseScore nicht gefunden. Kontrollieren Sie den Pfad! '
  }
}
```

```
# MuseScore aufrufen, Argumente übergeben und auf Abschluss warten:
# (ACHTUNG: MuseScore-Parameter sind case-sensitive, unterscheiden also
#  zwischen Groß- und Kleinschreibung! Der richtige Parameter lautet -o
#  wie in "Otto" und muss kleingeschrieben sein!)
Start-Process -Wait -FilePath $muse -ArgumentList '-o', $PdfFile, $MidiFile

# PDF-Datei öffnen (PDF-Viewer nötig):
Invoke-Item -Path $PdfFile
```

Listing 2.25: Automatisiert eine .mid-Datei in ein Notenblatt im PDF-Format konvertieren

Das Ergebnis ist eine PDF-Datei mit der gewünschten Partitur, die in Ihrem Standard-PDF-Viewer angezeigt wird (siehe Abbildung 2.23).

Erstaunlich, oder? Im Internet finden Sie unzählige kostenlose *.mid*-Dateien, und Sie könnten sich nun mit PowerShell daraus in wenigen Sekunden wertvolle Notensammlungen generieren.

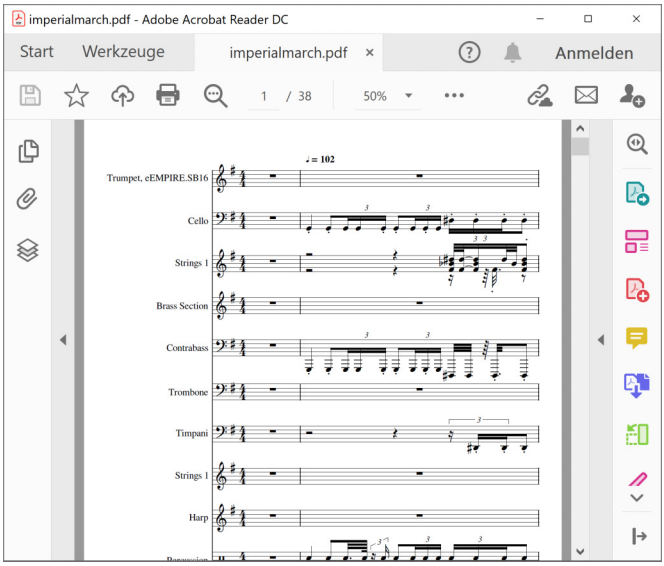

Abbildung 2.23: Automatisch aus .mid-Datei hergestellte Partitur als PDF-Datei

Schauen wir uns noch kurz an, wie Listing 2.25 diese Umwandlung bewerkstelligt hat.

Wieder war PowerShell nur »Dirigent« und hat die speziellen Fähigkeiten einzelner Befehle lediglich sinnvoll koordiniert:

- **Speicherorte festlegen:** Zuerst hat `Join-Path` die Pfade zusammengestellt, unter denen die Original-MIDI-Datei und die neue PDF-Version davon gespeichert werden sollen.
- **MIDI-Datei herunterladen:** Danach hat der Internetspezialist `Invoke-RestMethod` die MIDI-Datei aus dem Internet heruntergeladen. Hier könnten Sie also ebenso gut auch andere MIDI-Dateien herunterladen (wenn Sie deren URL kennen).
- **MuseScore finden:** Weil sich die Application *MuseScore* unter verschiedenen Namen an unterschiedlichen Orten befinden kann, testet `Test-Path` die beiden am häufigsten verwendeten Orte und Namen. Nur wenn an beiden Orten keine Application gefunden wird, löst `throw` einen Fehler aus und fordert, dass Sie sich der Sache annehmen und den Pfad zu *MuseScore* korrigieren (oder die Anwendung überhaupt erst installieren).