Inhaltsverzeichnis

PowerShell meistern — Kompendium

Dieses Kompendium bündelt alle Kapitel in einem durchgehenden Dokument.
Es ist als Nachschlagewerk für Einsteiger:innen und Fortgeschrittene gedacht, die PowerShell im Alltag sicher und effizient einsetzen wollen.

Aufbau und Ziel

- **Praxisorientiert**: Jeder Abschnitt enthält sofort anwendbare Beispiele.
- **Systematisch**: Von den Grundlagen über typische Workflows bis hin zu erweiterten Szenarien.
- **Erweiterbar**: Die Kapitel können beliebig ergänzt oder aktualisiert werden.

Wie dieses Dokument gelesen werden kann

- **Von vorne nach hinten** als kompletter Kurs.
- **Kapitelweise** als Nachschlagewerk zu bestimmten Themen.

- **Interaktiv**: Viele Beispiele lassen sich direkt in einer PowerShell-Sitzung ausprobieren.

  Hinweis: Dieses Dokument ersetzt keine offizielle Microsoft-Dokumentation.
  Es versteht sich als praxisnaher Leitfaden, der typische Aufgaben bündelt und anwendbar macht.

1. Einführung & Überblick

Dieses Kompendium führt Schritt für Schritt in PowerShell ein – von den Grundlagen bis hin zu komplexen Administrationsthemen. Die Kapitel sind so aufgebaut, dass sie logisch aufeinander aufbauen und ein roter Faden entsteht.

PowerShell ist mehr als nur eine Skriptsprache – es ist eine **Automatisierungs- und Verwaltungsplattform**. Jeder Themenblock im Kompendium zeigt, wie sich PowerShell als zentrales Werkzeug einsetzen lässt:

- **Grundlagen**: schaffen das Fundament, um Cmdlets, Variablen und Operatoren sicher zu nutzen.
- **Sprachelemente**: erklären, wie PowerShell als Programmiersprache funktioniert und wie du Logik umsetzt.
- **Objekte & Pipeline**: zeigen die besondere Stärke von PowerShell – alles sind Objekte, die flexibel weitergereicht werden können.
- **Fehlerbehandlung & Debugging**: sorgen für stabile Skripte, die auch in produktiven Umgebungen zuverlässig laufen.
- **Module & Automatisierung**: machen PowerShell skalierbar – von wiederverwendbaren Modulen bis hin zu CI/CD-Pipelines.
- **Administration**: PowerShell als Werkzeugkasten für IT-Admins – von Datei- und Registryverwaltung über Active Directory bis zu Exchange und Windows Management.
- **Cross-Plattform & Integration**: zeigen, wie PowerShell über Windows hinaus funktioniert und .NET oder GUIs integriert.
- **Sicherheit & Best Practices**: helfen, PowerShell verantwortungsvoll und sicher einzusetzen.

Damit wird klar: PowerShell ist nicht nur eine Sammlung von Befehlen, sondern ein **ganzheitliches Framework für Administration, Automatisierung und Entwicklung**.

1.1 Überblick als Strukturbaum

- Grundlagen
  - Tools & Hilfe-System
  - Datentypen & Variablen
  - Operatoren
- Sprachelemente
  - Bedingungen
  - Schleifen
  - Funktionen
  - Skripte
- Objekte & Pipeline
  - Objekte verstehen
  - Pipeline

- Fehler & Debugging
  - Fehlerbehandlung
  - Debugging
- Module & Automatisierung
  - Module & Funktionen
  - Remoting & Jobs
  - Paketmanagement & Gallery
  - CI/CD & Scheduler
  - Tests mit Pester
- Administration
  - Dateien & Registry
  - Prozesse & Netzwerk
  - Active Directory & Exchange
  - Windows Management
  - JEA
- Cross-Plattform & Integration
  - Linux & Mac
  - .NET-Integration
  - GUI-Tools
- Sicherheit & Standards
  - Signaturen
  - Profile & PSReadLine
  - Best Practices

1.2 Voraussetzungen und Setup (kurz)

- PowerShell 7.5 oder neuer
- Visual Studio Code mit PowerShell-Extension.

Version prüfen:

```
$PSVersionTable.PSVersion
code --version
```

2. Tools für PowerShell

Tools die das arbeiten mit bzw. in der PowerShell erleichtern.

2.1 Vorbereitung

Installiere die Schriftart Cascadia Code[1].
Sie ist optimiert für Programmierung und sorgt für bessere Lesbarkeit in PowerShell & VSCode.

2.2 PowerShell-Konsole

Starten

```
Start-Process "C:\Program Files\PowerShell\7\pwsh.exe"
```

[1] https://github.com/microsoft/cascadia-code

Tipp: Nutze Windows Terminal[2], um PowerShell, CMD & Bash in Tabs oder Splits zu verwenden.

Tastaturbefehle

Tastaturbefehle	Beschreibung
TAB	Befehlszeilenergänzung
STRG + C	Abbruch (oder Kopieren ab v5.0)
PFEIL-OBEN/-UNTEN	Blättert im Befehls-Cache
MARKIERUNG + ENTER	Kopiert die Markierung in die Zwischenablage
RECHTS-KLICK	Fügt die Zwischenablage ein
STRG + V	Einfügen (>= 5.0)
STRG + SPACE	Vorschlagsliste & Autovervollständigung für Parameter und Argumente

History verwalten

```
Get-History
$MaximumHistoryCount = 4096 # Achtung: Default-Wert, nicht reduzieren!
```

Konsolen-Logging (Transcript)

```
Start-Transcript C:\Temp\ps.log
## ... Befehle ausführen ...
Stop-Transcript
Get-Content C:\Temp\ps.log
```

Anzeigeoptionen anpassen

```
$FormatEnumerationLimit = -1  # unbegrenzt statt Default=4
```

2.3 Script Analyzer

Code-Check

```
Invoke-ScriptAnalyzer -Path .\script.ps1
Get-ScriptAnalyzerRule

## Formatieren
Invoke-Formatter -ScriptDefinition $definition
```

In VSCode sind Regelverstöße direkt im *Problems*-Fenster sichtbar.

2.4 Visual Studio Code

Empfohlen als Editor (Nachfolger der ISE)

[2] https://github.com/microsoft/terminal

Obligatorische Extensions

Titel	ID	Beschreibung
PowerShell	ms-vscode.powershell	PowerShell (.PS1) Integration
Better Comments	aaron-bond.better-comments	Kommentare farblich hervorheben
Code Spell Checker	streetsidesoftware.code-spell-checker + streetsidesoftware.code-spell-checker-german	Rechtschreibkorrektur (EN/DE)

Optionale Extensions

Titel	ID	Beschreibung
Markdown All in One	yzhang.markdown-all-in-one	Markdown (.md) Integration
Markdown PDF	yzane.markdown-pdf	Markdown ? PDF Export
MarkdownLint	davidanson.vscode-markdownlint	Markdown Style-Checking
XML Tools	dotjoshjohnson.xml	XML Integration
Vscode Google Translate	funkyremi.vscode-google-translate	Übersetzungstool
Pester	ps-pester.pester	Framework für Unit-Tests in PowerShell

Visual Studio Code Tastaturbefehle

Beschreibung	Tastenkürzel
Zeile/Selektion ausführen	F8
Online-Hilfe zum Cmdlet	CTRL+F1
Alle Regionen einklappen (PS1)	CTRL+K, CTRL+8
Alles einklappen (MD)	CTRL+K, CTRL+0
Alles aufklappen	CTRL+K, CTRL+J
Debugger starten (PS1 ausführen)	F5
Autovervollständigung öffnen	CTRL+SPACE
Dateien schnell öffnen	CTRL+P
Zeile nach unten verschieben	ALT+DOWN
Zeile(n) ein-/auskommentieren	CTRL+#

Alle Keybindings sind in `keybindings.json` anpassbar.

2.5 Windows Terminal

Features

- Mehrere Shells in Tabs & Splits

- Transparenz, Themes, Copy/Paste
- Download: Microsoft Store[3] oder GitHub[4]

Windows Terminal Tastaturbefehle

Beschreibung	Kürzel
Suchen	CTRL+SHIFT+F
Neuer vertikaler Bereich	ALT+SHIFT++
Neuer horizontaler Bereich	ALT+SHIFT+-
Bereich wechseln	ALT+PFEILTASTEN
Bereich schließen	CTRL+SHIFT+W

2.7 PowerShell ISE (Legacy)

Nicht mehr aktiv entwickelt, nur noch für alte Skripte relevant.

Empfehlung: **VSCode mit PowerShell-Extension**.

2.8 Weitere Tools

- CryptPad[5] ? kollaboratives Arbeiten
- RegEx101[6] ? Regex testen

3. Hilfe-System

PowerShell hat ein integriertes Hilfesystem.
Es liefert Dokumentation zu Cmdlets, Parametern und Konzepten (`about_*`).
Die Hilfe kann aktualisiert und auch offline gespeichert werden.

3.1 Hilfe aufrufen

```powershell
## Hilfe zu Cmdlets anzeigen
Get-Help Get-Command

## Varianten
Get-Help Get-Command -Detailed       # Details
Get-Help Get-Command -Examples       # Beispiele
Get-Help Get-Command -Full           # Volltext
Get-Help Get-Command -Online         # Doku im Browser
Get-Help Get-Command -Parameter Name # Nur Parameterinfo
```

[3] https://aka.ms/terminal
[4] https://github.com/microsoft/terminal
[5] https://cryptpad.fr
[6] https://regex101.com

3.2 Hilfe aktualisieren

```
## Hilfe aktualisieren
Update-Help -Force

## Hilfe zentral speichern & verteilen
Save-Help -DestinationPath C:\temp\Help
Update-Help -SourcePath C:\temp\Help
```

3.3 Hilfe durchsuchen

```
## Hilfe nach Cmdlets durchsuchen
Get-Help *service*

## Hilfe-Themen anzeigen
Get-Help about_*
```

3.4 Hilfe mit Get-Command nutzen

```
## Cmdlets nach Verb durchsuchen
Get-Command -Verb Get

## Cmdlets nach Noun durchsuchen
Get-Command -Noun Service

## Nach Modul suchen
Get-Command -Module NetTCPIP

## Parameter suchen
Get-Command -ParameterName ComputerName
```

`gcm` ist der Alias für `Get-Command`.

3.5 About-Themen

PowerShell enthält zusätzliche **Hilfethemen** zu Sprache, Syntax und Konzepten (`about_*`).

```
## Alle About-Themen anzeigen
Get-Help about_*


## Beispiel: Hilfe zu Arrays
Get-Help about_Arrays
```

3.6 Externe Hilfequellen

Neben der integrierten Hilfe sind auch **externe Quellen** nützlich:

- Microsoft Docs[7]

[7] https://learn.microsoft.com/powershell/

- PowerShell Gallery[8]
- Community (z. B. Blogs, GitHub, Foren)

4. Datentypen

In PowerShell hat jeder Wert einen Datentyp.
Variablen sind dynamisch, können aber mit **Typ-Literalen** festgelegt werden.

4.1 Einführung

```
$a = 5
$a.GetType()        # System.Int32

$a = "Text"
$a.GetType()        # System.String

[string]$b = 42
$b.GetType()        # System.String
```

4.2 Wichtige Standard-Datentypen

- Ganzzahl (Int32)
- Gleitkommazahl (Double)
- Zeichenkette (String)
- Wahrheitswert (Boolean)
- Datum/Zeit (DateTime)

```
[int]$zahl    = 42        # Ganzzahl
[string]$text = "Hallo"   # Zeichenkette
[bool]$wahr   = $true     # Wahrheitswert
[datetime]$d  = Get-Date  # Datum/Zeit
```

4.3 Typumwandlungen

```
[int]"42"             # String → Ganzzahl
[string]123           # Zahl → String
[datetime]"2025-01-01"  # String → Datum

"123".ToString()      # Zahl/String → Text
[bool]0               # False
[bool]1               # True
```

Manche Umwandlungen schlagen fehl:

[8] https://www.powershellgallery.com

```
[int]"abc"            # Fehler
```

4.4 Arrays und Hashtables

```
## Array
$a = @(1,2,3)
$a[0]         # Zugriff erstes Element → 1
$a.Length     # Länge → 3

## Hashtable
$h = @{Name="Attila"; Ort="Würzburg"}
$h["Name"]    # Zugriff über Schlüssel → Attila
$h.Ort        # Zugriff über Property → Würzburg
```

4.5 Objekte und Eigenschaften/Methoden

```
$p = Get-Process -Id $PID

## Eigenschaften
$p.ProcessName    # Prozessname
$p.Id             # Prozess-ID

## Methoden
$p.Kill()         # Prozess beenden
"Hallo".ToUpper() # String in Großbuchstaben
```

4.6 Besonderheiten von Null und `$null`

`$null` steht für **kein Wert**.
Best Practice: `$null` immer links im Vergleich schreiben.

```
$a = $null
$null -eq $a      # True
$a -eq $null      # True (funktioniert, aber schlechter Stil)

$a = ""
$null -eq $a      # False (leer -ne null)
```

4.7 Operatoren für Typprüfung und Casting

```
"Text" -is [string]    # True
42 -is [int]           # True
42 -is [string]        # False

"42" -as [int]         # 42
"abc" -as [int]        # $null (fehlgeschlagen)
```

4.8 Vergleichsoperatoren

```
5 -eq 5     # Gleich
5 -ne 4     # Ungleich
5 -gt 3     # Größer als
5 -lt 10    # Kleiner als

"Test" -eq "test"    # True (nicht case-sensitiv)
"Test" -ceq "test"   # False (case-sensitiv)
```

4.9 Logische Operatoren

```
$true  -and $false    # False
$true  -or  $false    # True
-not $true            # False
```

4.10 Collections und Enumerables

```
## Liste mit mehreren Werten
$list = 1..5

foreach ($i in $list) {
    $i * 2  # Ausgabe: 2,4,6,8,10
}
```

4.11 Zeichenketten (Strings)

```
$name = "Attila"

## Verkettung
"Hallo " + $name      # Hallo Attila

## Interpolation
"Hallo $name"         # Hallo Attila

## Länge
$name.Length           # 6
```

4.12 Escape-Sequenzen

```
"Zeile1`nZeile2"    # Neue Zeile
"Tab`tgetrennt"     # Tabulator
```

4.13 Reguläre Ausdrücke (Regex)

```
"abc123" -match "[a-z]+[0-9]+"    # True
```

```
$matches[0]    # abc123
```

4.14 Zahlenformate

```
## Zahl formatieren
$zahl = 1234.56

"{0:N2}" -f $zahl    # 2.234,56 (2 Nachkommastellen)
"{0:C}" -f $zahl     # Währungsformat
```

5. Variablen

In PowerShell werden Variablen mit $ eingeleitet. Sie können Werte unterschiedlichen Typs speichern und sind standardmäßig dynamisch.

5.1 Variablen erstellen und verwenden

```
$a = 5
$b = "Hallo"
$c = Get-Date
```

5.2 Variablen-Typ festlegen

```
[int]$zahl = 42
[string]$text = "Text"
[datetime]$datum = "2025-01-01"
```

5.3 Besondere Variablen

- `$null` → Kein Wert
- `$_` → aktuelles Pipeline-Objekt
- `$?` → Status des letzten Befehls (True/False)
- `$LASTEXITCODE` → Exitcode des letzten nativen Programms
- `$PID` → Prozess-ID der aktuellen PowerShell-Instanz
- `$PSVersionTable` → Version und Umgebung

```
$PSVersionTable.PSVersion
```

5.4 Variablenbereich (Scope)

Variablen können in unterschiedlichen **Scopes** existieren: Global, Script, Local.

```
$global:x = "Global"
$script:y = "Script"
$local:z  = "Local"
```

Best Practice: Nur Scopes setzen, wenn wirklich nötig – sonst Standard-Variablen nutzen.

6. Operatoren

PowerShell bietet eine Vielzahl von Operatoren für Vergleiche, Berechnungen und Logik.

6.1 Arithmetische Operatoren

```
5 + 3   # Addition → 8
5 - 3   # Subtraktion → 2
5 * 3   # Multiplikation → 15
5 / 3   # Division → 1,666...
5 % 3   # Modulo → 2
```

6.2 Vergleichsoperatoren

```
5 -eq 5    # Gleich → True
5 -ne 4    # Ungleich → True
5 -gt 3    # Größer als → True
5 -lt 10   # Kleiner als → True

"Test" -eq "test"   # True (nicht case-sensitiv)
"Test" -ceq "test"  # False (case-sensitiv)
```

6.3 Logische Operatoren

```
$true  -and $false   # False
$true  -or  $false   # True
-not $true           # False
```

6.4 Zuweisungsoperatoren

```
$a = 5
$a += 3    # 8
$a -= 2    # 6
$a *= 4    # 24
$a /= 6    # 4
```

Best Practice: Zuweisungsoperatoren sparsam nutzen – sie verkürzen Code, können aber die Lesbarkeit mindern.

7. Bedingungen

Mit Bedingungen steuerst du den Ablauf von PowerShell-Skripten.

7.1 If-Bedingungen

```
$a = 5
if ($a -gt 3) {
    "a ist größer als 3"
}
```

7.2 If-Else

```
$a = 2
if ($a -gt 3) {
    "a ist größer als 3"
}
else {
    "a ist kleiner/gleich 3"
}
```

7.3 ElseIf-Ketten

```
$a = 3
if ($a -gt 5) {
    "a > 5"
}
elseif ($a -eq 3) {
    "a = 3"
}
else {
    "a ist kleiner als 5 und nicht 3"
}
```

Weitere Infos:

```
Get-Help -Name about_If -ShowWindow
```

7.4 Switch

```
$wert = "B"
switch ($wert) {
    "A" { "Wert ist A" }
    "B" { "Wert ist B" }
    default { "Unbekannt" }
}
```

Weitere Infos:

```
Get-Help -Name about_Switch -ShowWindow
```

Best Practice: Switch nutzen, wenn viele Vergleiche nötig sind – übersichtlicher als viele `elseif`.

8. Schleifen

Schleifen wiederholen Anweisungen, bis eine Bedingung erfüllt ist oder eine Menge von Objekten abgearbeitet wurde.

8.1 For-Schleife

Eine **for-Schleife** eignet sich, wenn die Anzahl der Durchläufe bekannt ist.

```
for ($i = 1; $i -le 5; $i++) {
    "Wert: $i"
}
```

Weitere Infos:

```
Get-Help -Name about_For -ShowWindow
```

8.2 Foreach-Schleife

Mit **foreach** iterierst du direkt über alle Elemente einer Sammlung oder Pipeline.

```
$liste = @("A","B","C")
foreach ($element in $liste) {
    "Element: $element"
}
```

Weitere Infos:

```
Get-Help -Name about_Foreach -ShowWindow
```

8.3 While-Schleife

Eine **while-Schleife** läuft so lange, wie die Bedingung wahr ist.

```
$i = 1
while ($i -le 3) {
    "Wert: $i"
    $i++
}
```

Weitere Infos:

```
Get-Help -Name about_While -ShowWindow
```

8.4 Do-While-Schleife

Die **do-while-Schleife** führt den Block mindestens einmal aus und prüft die Bedingung danach.

```
$i = 1
do {
    "Wert: $i"
    $i++
} while ($i -le 3)
```

Weitere Infos:

```
Get-Help -Name about_Do -ShowWindow
```

8.5 Do-Until-Schleife

Die **do-until-Schleife** läuft mindestens einmal und wiederholt sich, bis die Bedingung erfüllt ist.

```
$i = 1
do {
    "Wert: $i"
    $i++
} until ($i -gt 3)
```

Weitere Infos:

```
Get-Help -Name about_Do -ShowWindow
```

8.6 Break und Continue

Mit **break** wird eine Schleife beendet, mit **continue** nur der aktuelle Durchlauf übersprungen.

```
for ($i = 1; $i -le 5; $i++) {
    if ($i -eq 3) { continue }  # überspringt 3
    if ($i -eq 5) { break }     # beendet Schleife
    "Wert: $i"
}
```

Weitere Infos:

```
Get-Help -Name about_Break -ShowWindow
Get-Help -Name about_Continue -ShowWindow
```

Best Practice: `foreach` bevorzugen, wenn Objekte direkt aus der Pipeline verarbeitet werden – übersichtlicher und sicherer.

9. Funktionen

Funktionen ermöglichen es, wiederverwendbare Codeblöcke zu definieren und übersichtlicher zu arbeiten.

9.1 Einfache Funktionen

Eine Funktion wird mit dem Schlüsselwort **function** definiert.

```
function Hallo {
    "Hallo Welt"
}
```

```
Hallo
```

Weitere Infos:

```
Get-Help -Name about_Functions -ShowWindow
```

9.2 Funktionen mit Parametern

Funktionen können Eingaben über **Parameter** erhalten.

```
function Begruessung {
    param($Name)
    "Hallo $Name"
}

Begruessung -Name "Attila"
```

Weitere Infos:

```
Get-Help -Name about_Parameters -ShowWindow
```

9.3 Rückgabewerte

Funktionen geben standardmäßig alles zurück, was sie ausgeben.

```
function Quadrat {
    param($Zahl)
    return $Zahl * $Zahl
}

Quadrat -Zahl 4   # 16
```

Weitere Infos:

```
Get-Help -Name about_Return -ShowWindow
```

9.4 Erweiterte Funktionen

Erweiterte Funktionen verhalten sich wie Cmdlets und bieten Features wie Parameterattribute und Pipeline-Unterstützung.

```
function Get-Quadrat {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        [int]$Zahl
    )
    process {
        $Zahl * $Zahl
    }
}
```

```
1..5 | Get-Quadrat
```

Weitere Infos:

```
Get-Help -Name about_Functions_Advanced -ShowWindow
```

Best Practice: Für wiederverwendbaren Code immer Funktionen schreiben – mit klaren Parameternamen und nach Möglichkeit Pipeline-Unterstützung.

10. Module

Module bündeln Funktionen, Cmdlets und Ressourcen und machen sie wiederverwendbar.

10.1 Module laden

Ein Modul kann mit **Import-Module** geladen werden.

```
Import-Module Microsoft.PowerShell.Management
```

Weitere Infos:

```
Get-Help -Name Import-Module -ShowWindow
```

10.2 Installieren von Modulen

Neue Module können aus der PowerShell Gallery installiert werden.

```
Install-Module -Name Pester -Scope CurrentUser
```

Weitere Infos:

```
Get-Help -Name Install-Module -ShowWindow
```

10.3 Auflisten installierter Module

```
Get-Module -ListAvailable
```

Weitere Infos:

```
Get-Help -Name Get-Module -ShowWindow
```

10.4 Export von Modulen

Eigene Module können in `.psm1`-Dateien gespeichert werden.

```
function Hallo {
    "Hallo Welt"
}

Export-ModuleMember -Function Hallo
```

Weitere Infos:

```
Get-Help -Name Export-ModuleMember -ShowWindow
```

Best Practice: Eigene Funktionen in Modulen organisieren – erleichtert Wiederverwendung und Verteilung.

11. Skripte

Skripte sind PowerShell-Dateien mit der Endung `.ps1`, die mehrere Befehle enthalten und automatisierte Abläufe ermöglichen.

11.1 Skript erstellen

Ein Skript wird als Textdatei mit der Endung `.ps1` gespeichert.

```
## Datei: Hallo.ps1
Write-Output "Hallo Welt"
```

Ausführen:

```
./Hallo.ps1
```

Weitere Infos:

```
Get-Help -Name about_Scripts -ShowWindow
```

11.2 Ausführungsrichtlinien

Die **Execution Policy** steuert, ob Skripte ausgeführt werden dürfen.

```
Get-ExecutionPolicy
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

Weitere Infos:

```
Get-Help -Name about_Execution_Policies -ShowWindow
```

11.3 Parameter in Skripten

Skripte können Eingaben über **Param-Block** erhalten.

```
param(
    [string]$Name
)

Write-Output "Hallo $Name"
```

Aufruf:

```
./Hallo.ps1 -Name Attila
```

Weitere Infos:

```
Get-Help -Name about_Parameters -ShowWindow
```

11.4 Rückgabewerte

Skripte geben standardmäßig alle Ausgaben zurück.

```
## Datei: Quadrat.ps1
param([int]$Zahl)

$Zahl * $Zahl
```

Aufruf:

```
./Quadrat.ps1 -Zahl 5   # 25
```

Best Practice: Skripte immer mit Parametern schreiben und keine festen Werte im Code verwenden – macht sie wiederverwendbar.

12. Pipeline

Die Pipeline (|) verbindet Befehle, sodass die Ausgabe des einen als Eingabe für den nächsten dient.

12.1 Grundprinzip

```
Get-Process | Sort-Object CPU -Descending
```

Weitere Infos:

```
Get-Help -Name about_Pipelines -ShowWindow
```

12.2 Mehrstufige Pipeline

```
Get-Process |
    Where-Object CPU -gt 100 |
    Sort-Object CPU -Descending |
    Select-Object -First 5
```

Weitere Infos:

```
Get-Help -Name about_Where -ShowWindow
```

12.3 Pipeline und Formatierung

```
Get-Service | Format-Table Name, Status
```

Weitere Infos:

```
Get-Help -Name about_Format.ps1xml -ShowWindow
```

12.4 Pipeline und Export

```
Get-Process | Export-Csv -Path Prozesse.csv -NoTypeInformation
```

Weitere Infos:

```
Get-Help -Name Export-Csv -ShowWindow
```

Best Practice: In Pipelines immer zuerst filtern (`Where-Object`), dann sortieren (`Sort-Object`), und zuletzt formatieren oder exportieren – für bessere Performance und Übersicht.

13. Objekte

In PowerShell ist fast alles ein Objekt. Objekte haben **Eigenschaften** (Daten) und **Methoden** (Funktionen).

13.1 Eigenschaften abfragen

```
Get-Process | Select-Object -Property Name, Id
```

Weitere Infos:

```
Get-Help -Name Select-Object -ShowWindow
```

13.2 Methoden verwenden

```
"hallo".ToUpper()
```

Weitere Infos:

```
Get-Help -Name about_Methods -ShowWindow
```

13.3 Objekte filtern

```
Get-Service | Where-Object Status -eq "Running"
```

Weitere Infos:

```
Get-Help -Name Where-Object -ShowWindow
```

13.4 Objekte sortieren

```
Get-Process | Sort-Object CPU -Descending
```

Weitere Infos:

```
Get-Help -Name Sort-Object -ShowWindow
```

13.5 Objekte formatieren

```
Get-Service | Format-Table Name, Status
```

Weitere Infos:

```
Get-Help -Name about_Format.ps1xml -ShowWindow
```

Best Practice: Erst filtern und sortieren, dann formatieren – so bleibt die Verarbeitung performant und übersichtlich.

14. Fehler und Ausnahmen

PowerShell unterscheidet zwischen **Fehlern (non-terminating)**, die das Skript weiterlaufen lassen, und **Ausnahmen (terminating)**, die den Ablauf sofort beenden. Mit Parametern und Präferenz-Variablen steuerst du, wie damit umgegangen wird.

```
## Kompakte Fehleransicht (PS 7+)
$ErrorView = 'ConciseView'

## Details zum letzten Fehler anzeigen
Get-Error -Newest 1

## Fehler sofort abbrechen lassen
Get-ChildItem C:\NoDir -ErrorAction Stop
```

Weitere Infos:

```
Get-Help -Name 'about_Preference_Variables' -ShowWindow
```

14.1 Meldungen unterdrücken

Fehlermeldungen und Ausgaben lassen sich lokal pro Befehl oder global per Variablensteuerung unterdrücken. Das macht Skripte robuster und verhindert unnötige Log-Ausgaben.

```
## Lokale Steuerung
Get-Process -FileVersionInfo `
  -ErrorAction Stop `              # Fehler als Ausnahme
  -WarningAction Ignore `          # Warnungen ausblenden
  -InformationAction Continue `    # Infos anzeigen
  -Verbose                         # ausführliche Meldungen
```

```
## Globale Voreinstellungen sichern, ändern und zurücksetzen
$bak = [ordered]@{
  ErrorAction    = $ErrorActionPreference
  Warning        = $WarningPreference
  Information    = $InformationPreference
  Verbose        = $VerbosePreference
}
$ErrorActionPreference = 'Stop'
$WarningPreference     = 'Stop'
$InformationPreference = 'SilentlyContinue'
```

```
$VerbosePreference     = 'SilentlyContinue'
## ... Code ...
$ErrorActionPreference = $bak.ErrorAction
$WarningPreference     = $bak.Warning
$InformationPreference = $bak.Information
$VerbosePreference     = $bak.Verbose
```

```
## Ausgabe ins Leere schicken
Get-Process | Out-Null
Get-Process > $null
```

Weitere Infos:

```
Get-Help -Name 'about_CommonParameters' -ShowWindow
```

14.2 Fehler analysieren

Fehler können über die automatische Variable `$Error`, über `Get-Error` oder über `$?` und `$LASTEXITCODE` nachvollzogen werden. So erkennst du, ob ein Befehl erfolgreich war oder nicht.

```
## Fehlerhistorie
$Error[0] | Format-List *
"Fehleranzahl: $($Error.Count)"
$Error.Clear()
```

```
## Exception-Details auswerten
try { 1/0 } catch {
  $_.Exception.Message
  $_.Exception.GetType().FullName
  $_.Exception.InnerException
}
```

```
## Erfolg prüfen
if ($?) { 'OK' } else { 'Fehler' }

## Exitcode nativer Tools
ping.exe 127.0.0.1 > $null
$LASTEXITCODE
```

Weitere Infos:

```
Get-Help -Name 'about_Automatic_Variables' -ShowWindow
```

14.3 Fehler behandeln

Fehler lassen sich mit `try`, `catch` und `finally` abfangen. Dabei sollten spezifische Fehler zuerst behandelt und unspezifische zuletzt gefangen werden.

```
try {
  $bak = $ErrorActionPreference
```

```
  $ErrorActionPreference = 'Stop'
  Get-Content 'C:\Temp\wichtig.txt'
}
catch [System.Management.Automation.ItemNotFoundException] {
  New-Item 'C:\Temp\wichtig.txt' -ItemType File | Out-Null
}
catch [System.IO.IOException] {
  # Alternative Behandlung
}
catch {
  Write-Error -Message $_.Exception.Message -ErrorId 'Unhandled' -Category
  ↪  NotSpecified
  throw
}
finally {
  $ErrorActionPreference = $bak
}
```

Weitere Infos:

```
Get-Help -Name 'about_Try_Catch_Finally' -ShowWindow
```

14.4 Eigene Fehler auslösen

Eigene Fehler sind nützlich, wenn Eingaben nicht den Erwartungen entsprechen oder Vorbedingungen verletzt sind. So stellst du sicher, dass Fehler früh sichtbar werden.

```
## Einfache Ausnahme
throw "Ungültige Eingabe."

## Typisierte Ausnahme
throw [System.ArgumentException]::new("Wert ist ungültig", "Path")

## Nicht-terminierender Fehler
Write-Error -Message "Benutzer nicht gefunden" -Category ObjectNotFound
↪  -ErrorId 'User404' -TargetObject 'max.mustermann'
```

```
## Validate-Attribute prüfen Eingaben automatisch
param(
  [ValidateRange(1,10)] [int]$Level,
  [ValidatePattern('^[A-Z]{2,5}[0-9]+$')] [string]$HostName
)
```

Weitere Infos:

```
Get-Help -Name 'about_Throw' -ShowWindow
```

15. Debugging

Debugging ermöglicht es, PowerShell-Code gezielt zu unterbrechen und Schritt für Schritt zu prüfen. Damit lassen sich Fehlerquellen schneller finden und Abläufe besser nachvollzie-

hen. Typische Werkzeuge sind Breakpoints, der integrierte Debugger und die Debugging-Funktionen in VSCode.

```
Get-Help -Name 'about_Debuggers' -ShowWindow
```

15.1 Breakpoints setzen

Breakpoints stoppen den Code an einer bestimmten Stelle. So kannst du Variablen einsehen, Ausgaben überprüfen und den Ablauf kontrollieren.

```
## Breakpoint in Datei bei bestimmter Zeile
Set-PSBreakpoint -Script .\Skript.ps1 -Line 10

## Breakpoint auf Variable
Set-PSBreakpoint -Variable counter -Mode Write

## Breakpoint auf Funktion
Set-PSBreakpoint -Command Get-Process

## Breakpoints anzeigen und löschen
Get-PSBreakpoint
Remove-PSBreakpoint -Id 1
```

15.2 Debugger verwenden

Der integrierte Debugger führt Code schrittweise aus und zeigt, wie Variablen und Ausgaben entstehen. So kannst du komplexe Logik nachvollziehen und Fehler lokalisieren.

```
## Skript mit Debugger starten
powershell.exe -Command "Set-PSDebug -Step"

## Eingaben im Debugger
s  # step into
o  # step out
v  # step over
c  # continue
q  # quit
```

```
## Debug-Optionen prüfen und steuern
Get-PSDebug
Set-PSDebug -Trace 1    # Tracing aktivieren
Set-PSDebug -Trace 0    # Tracing deaktivieren
```

15.3 Debugging in VSCode

Mit der PowerShell-Extension in Visual Studio Code kannst du komfortabel debuggen. Breakpoints setzt du per Klick, und Variablen sowie Call-Stack sind im Debug-Panel sichtbar.

```
// launch.json Beispiel für Debug-Config
{
```

```
  "version": "0.2.0",
  "configurations": [
    {
      "type": "PowerShell",
      "request": "launch",
      "name": "PowerShell Debug",
      "script": "${file}"
    }
  ]
}
```

15.4 Remote-Debugging

Manche Fehler treten nur in der Zielumgebung auf. Mit Remote-Debugging kannst du Prozesse auf anderen Maschinen untersuchen und Skripte direkt dort nachverfolgen.

```
## Remote-Session starten und Runspace debuggen
$session = New-PSSession -ComputerName Server01
Enter-PSHostProcess -Id (Get-Process -Name pwsh).Id -AppDomainName
↪  DefaultAppDomain
Debug-Runspace -Id 1
```

15.5 Tipps & Best Practices

Effizientes Debugging spart Zeit und reduziert Fehlersuchen:

- Breakpoints gezielt setzen, nicht flächendeckend.
- Tracing nur bei Bedarf nutzen, da es Leistung kostet.
- In VSCode Call-Stack und Variablenüberwachung aktiv einsetzen.
- Remote-Debugging nur über sichere Sessions durchführen.
- Zum Schluss alle Breakpoints entfernen, um saubere Skripte zu gewährleisten.

```
Get-PSBreakpoint | Remove-PSBreakpoint
```

16. Module und Funktionen

Module und Funktionen helfen dabei, PowerShell-Code zu strukturieren, wiederverwendbar zu machen und in Projekte zu integrieren. Während Funktionen kleine Codeblöcke abbilden, bündeln Module mehrere Funktionen oder Cmdlets in einer Einheit.

```
Get-Help -Name 'about_Modules' -ShowWindow
```

16.1 Funktionen erstellen

Funktionen sind benannte Codeblöcke, die wiederholt aufgerufen werden können. Sie erleichtern Wartung und Lesbarkeit.

```
## Einfache Funktion
function Hallo {
    "Hallo Welt"
}
```

```
Hallo
```

Weitere Infos:

```
Get-Help -Name 'about_Functions' -ShowWindow
```

16.2 Funktionen mit Parametern

Funktionen können Eingaben über Parameter entgegennehmen. So lassen sich flexible und wiederverwendbare Abläufe entwickeln.

```
function Begruessung {
    param($Name)
    "Hallo $Name"
}

Begruessung -Name "Attila"
```

Weitere Infos:

```
Get-Help -Name 'about_Parameters' -ShowWindow
```

16.3 Erweiterte Funktionen

Erweiterte Funktionen verhalten sich wie Cmdlets. Mit `CmdletBinding()` und Parametern erhält man zusätzliche Features wie Pipeline-Support und Validierungen.

```
function Get-Quadrat {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory)] [int]$Zahl
    )
    $Zahl * $Zahl
}

Get-Quadrat -Zahl 5
```

Weitere Infos:

```
Get-Help -Name 'about_Functions_Advanced' -ShowWindow
```

16.4 Module erstellen

Module sind Sammlungen von Funktionen und Cmdlets in einer Datei oder einem Ordner. Mit ihnen kannst du Code sauber kapseln und weitergeben.

```
## Modul erstellen
New-Item -ItemType Directory -Path .\MeineModule
"function Hallo { 'Hallo aus Modul' }" | Out-File .\MeineModule\Hallo.psm1

## Modul importieren
```

```
Import-Module .\MeineModule\Hallo.psm1

## Funktion aus Modul nutzen
Hallo
```

Weitere Infos:

```
Get-Help -Name 'about_Modules' -ShowWindow
```

16.5 Module verwalten

PowerShell bietet Cmdlets, um Module zu finden, installieren, laden und zu entfernen. So können Bibliotheken zentral gepflegt werden.

```
## Modul aus PSGallery installieren
Install-Module -Name Pester -Scope CurrentUser

## Modul laden
Import-Module Pester

## Verfügbare Module anzeigen
Get-Module -ListAvailable

## Modul entfernen
Remove-Module Pester
```

Weitere Infos:

```
Get-Help -Name 'about_Module' -ShowWindow
```

17. PowerShell Remoting

PowerShell Remoting ermöglicht es, Befehle und Skripte auf entfernten Computern auszuführen. Damit lassen sich Systeme zentral verwalten und Aufgaben automatisieren, ohne sich lokal anmelden zu müssen.

```
Get-Help -Name 'about_Remote' -ShowWindow
```

17.1 Grundlagen

Remoting basiert auf WS-Man (WinRM) oder SSH. Standardmäßig ist WS-Man auf Windows verfügbar, während SSH plattformübergreifend genutzt werden kann.

```
## WinRM für Remoting aktivieren (nur einmal pro Zielsystem nötig)
Enable-PSRemoting -Force

## Remote-Sitzung erstellen
Enter-PSSession -ComputerName Server01

## Remote-Befehl ausführen
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process }
```

Weitere Infos:

```
Get-Help -Name 'about_Remote' -ShowWindow
```

17.2 Sitzungen verwalten

Mit Sitzungen (`PSSession`) lassen sich mehrere Remote-Verbindungen parallel verwalten. Das spart Zeit und Ressourcen.

```
## Neue Sitzung erstellen
$session = New-PSSession -ComputerName Server01

## Befehl in Sitzung ausführen
Invoke-Command -Session $session -ScriptBlock { Get-Service }

## Alle Sitzungen anzeigen
Get-PSSession

## Sitzung schließen
Remove-PSSession $session
```

Weitere Infos:

```
Get-Help -Name 'about_PSSessions' -ShowWindow
```

17.3 Remoting über SSH

Ab PowerShell 7 lässt sich Remoting auch über SSH nutzen. Das ist besonders für Linux- oder gemischte Umgebungen hilfreich.

```
## Verbindung über SSH
Enter-PSSession -HostName server01.contoso.com -UserName admin

## Befehl über SSH ausführen
Invoke-Command -HostName server01.contoso.com -UserName admin -ScriptBlock
↪  { hostname }
```

Weitere Infos:

```
Get-Help -Name 'about_Remote' -ShowWindow
```

17.4 Befehle auf mehreren Systemen

Mit Remoting kannst du Befehle gleichzeitig auf mehreren Rechnern ausführen. So lassen sich Massen-Operationen einfach umsetzen.

```
## Gleichen Befehl auf mehreren Computern ausführen
Invoke-Command -ComputerName Server01,Server02,Server03 -ScriptBlock {
↪  Get-ComputerInfo }
```

17.5 Sicherheit und Best Practices

- Remoting nur über gesicherte Netzwerke verwenden.
- Authentifizierung per Kerberos (Domäne) oder Zertifikaten bevorzugen.
- Zugriff nur für Administratoren oder spezielle Servicekonten zulassen.
- Nach Nutzung Sitzungen schließen und nicht offen lassen.
- Für plattformübergreifende Szenarien SSH statt WinRM nutzen.

```
## Alle aktiven Remote-Sitzungen prüfen und beenden
Get-PSSession | Remove-PSSession
```

18. Jobs und parallele Ausführung

Mit Jobs lassen sich Aufgaben im Hintergrund oder parallel ausführen, ohne die aktuelle PowerShell-Sitzung zu blockieren. So können lange laufende Prozesse nebenbei laufen, während du weiterarbeitest.

```
Get-Help -Name 'about_Jobs' -ShowWindow
```

18.1 Hintergrundjobs

Hintergrundjobs starten Befehle asynchron. Das Ergebnis wird gespeichert und kann später abgefragt werden.

```
## Job starten
Start-Job -ScriptBlock { Get-Process }

## Aktive Jobs anzeigen
Get-Job

## Ergebnisse abrufen
Receive-Job -Id 1

## Jobs entfernen
Remove-Job -Id 1
```

Weitere Infos:

```
Get-Help -Name 'about_Jobs' -ShowWindow
```

18.2 Remoting-Jobs

Jobs können auch auf entfernten Computern laufen. So lassen sich mehrere Systeme parallel steuern.

```
## Job auf Remote-Computer starten
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service } -AsJob

## Alle Jobs anzeigen
Get-Job

## Ergebnis abrufen
```

```
Receive-Job -Id 2
```

Weitere Infos:

```
Get-Help -Name 'about_Remote_Jobs' -ShowWindow
```

18.3 ThreadJobs

ThreadJobs sind eine leichtere und schnellere Variante von Jobs, da sie direkt im gleichen Prozess laufen. Sie sind für viele parallele Tasks geeignet.

```
## ThreadJob starten (PS 7+)
Start-ThreadJob -ScriptBlock { Get-Date; Start-Sleep 3; Get-Date }

## Jobs überwachen
Get-Job

## Ergebnis abholen
Receive-Job -Id 3
```

Weitere Infos:

```
Get-Help -Name 'about_Thread_Jobs' -ShowWindow
```

18.4 ForEach-Object -Parallel

Mit PowerShell 7 lässt sich die Pipeline direkt parallelisieren. So können große Datenmengen effizient verarbeitet werden.

```
1..5 | ForEach-Object -Parallel {
    "Task $_ gestartet"
    Start-Sleep -Seconds 2
    "Task $_ fertig"
}
```

Weitere Infos:

```
Get-Help -Name 'ForEach-Object' -ShowWindow
```

18.5 Best Practices

- Für kurze, schnelle Tasks → **ThreadJobs** nutzen.
- Für komplexe, unabhängige Tasks → **Hintergrundjobs** verwenden.
- Für Remote-Systeme → **Remoting-Jobs** einsetzen.
- Ressourcen im Blick behalten: viele parallele Jobs können Systemlast stark erhöhen.
- Jobs nach Abschluss aufräumen, um Speicher zu sparen.

```
Get-Job | Remove-Job
```

19. Fehlerkultur & Best Practices

Eine saubere Fehlerkultur sorgt für robuste Skripte und erleichtert die Wartung. PowerShell bietet viele Mechanismen, um Fehler frühzeitig zu erkennen, gezielt zu behandeln und aussagekräftige Meldungen auszugeben.

19.1 Klare Fehlermeldungen

Aussagekräftige Fehlermeldungen helfen beim Debuggen und im Betrieb. Statt kryptischer Meldungen sollten sie Ursache und mögliche Lösung andeuten.

```
Write-Error -Message "Datei konnte nicht geladen werden: Pfad fehlt"
↪  -Category ObjectNotFound
```

19.2 Fehler frühzeitig prüfen

Vorbedingungen sollten geprüft werden, bevor kritischer Code ausgeführt wird. So lassen sich Fehler vermeiden, bevor sie entstehen.

```
if (-not (Test-Path "C:\\Temp\\config.json")) {
    throw "Konfigurationsdatei fehlt!"
}
```

19.3 Exceptions gezielt abfangen

Fehler sollten nur dann behandelt werden, wenn sie sinnvoll gelöst werden können. Unspezifisches Abfangen kann Fehler verschleiern.

```
try {
    Get-Content "C:\\Temp\\Daten.csv"
}
catch [System.IO.FileNotFoundException] {
    "Datei nicht gefunden - bitte prüfen."
}
```

19.4 Logging nutzen

Fehler und wichtige Ereignisse sollten protokolliert werden. So können Abläufe später nachvollzogen werden.

```
try {
    # Beispielcode
}
catch {
    Add-Content -Path "C:\\Logs\\script.log" -Value "Fehler:
    ↪  $($_.Exception.Message)"
}
```

19.5 Best Practices zusammengefasst

- Fehlermeldungen klar und eindeutig formulieren.
- Vorbedingungen prüfen, bevor Code ausgeführt wird.

- Exceptions gezielt abfangen und nur dort, wo sinnvoll.
- Logging einbauen, um Abläufe nachvollziehbar zu machen.
- Fehler nicht verschweigen – lieber sauber melden und ggf. abbrechen.

Weitere Infos:

```
Get-Help -Name 'about_Try_Catch_Finally' -ShowWindow
Get-Help -Name 'about_Throw' -ShowWindow
```

20. PowerShell und Dateien

PowerShell bietet zahlreiche Cmdlets zum Arbeiten mit Dateien und Verzeichnissen. Damit lassen sich Dateien lesen, schreiben, durchsuchen und verwalten.

```
Get-Help -Name 'about_Filesystem_Provider' -ShowWindow
```

20.1 Dateien lesen und schreiben

Dateien können direkt mit Cmdlets geöffnet und bearbeitet werden. So lassen sich Inhalte schnell einlesen oder speichern.

```
## Datei einlesen
Get-Content C:\\Temp\\info.txt

## Datei schreiben
"Hallo Welt" | Out-File C:\\Temp\\neu.txt

## Text anhängen
"Zusatz" | Add-Content C:\\Temp\\neu.txt
```

20.2 Dateien durchsuchen

Mit `Select-String` lassen sich Inhalte effizient durchsuchen, vergleichbar mit `grep`.

```
## Zeilen mit "Error" finden
Select-String -Path C:\\Temp\\*.log -Pattern "Error"

## Ergebnis enthält Datei, Zeilennummer und Text
```

20.3 Dateien verschieben und kopieren

Dateien können einfach kopiert, verschoben oder umbenannt werden.

```
## Datei kopieren
Copy-Item C:\\Temp\\neu.txt C:\\Backup\\neu.txt

## Datei verschieben
Move-Item C:\\Temp\\neu.txt C:\\Backup\\neu.txt

## Datei umbenennen
Rename-Item C:\\Backup\\neu.txt -NewName alt.txt
```

20.4 Dateiinformationen abrufen

Mit den Cmdlets lassen sich Metadaten wie Größe, Erstellungsdatum oder Attribute abfragen.

```
## Informationen zu Datei anzeigen
Get-Item C:\\Temp\\info.txt | Select-Object Name,Length,CreationTime
```

20.5 Dateien löschen

Dateien können direkt aus PowerShell entfernt werden. Vorsicht: Gelöschte Dateien landen nicht im Papierkorb.

```
## Datei löschen
Remove-Item C:\\Temp\\alt.txt

## Mehrere Dateien löschen
Remove-Item C:\\Temp\\*.bak
```

20.6 Best Practices

- Immer Pfade überprüfen, bevor Dateien gelöscht oder überschrieben werden.
- `-WhatIf` nutzen, um Änderungen zu simulieren.
- Für große Dateien `-ReadCount` bei `Get-Content` einsetzen.
- Bei Logdateien gezielt `Select-String` statt komplettes Einlesen nutzen.

```
## Sicheres Löschen testen
Remove-Item C:\\Temp\\*.log -WhatIf
```

21. Registry und Umgebungsvariablen

PowerShell erlaubt den direkten Zugriff auf Registry und Umgebungsvariablen. Damit lassen sich Systemeinstellungen abfragen und anpassen.

```
Get-Help -Name 'about_Registry' -ShowWindow
```

21.1 Registry abfragen

Die Registry wird wie ein Dateisystem behandelt. Mit `Get-Item` oder `Get-ChildItem` lassen sich Schlüssel und Werte anzeigen.

```
## Registry-Schlüssel anzeigen
Get-ChildItem HKLM:\\Software

## Einzelnen Wert auslesen
Get-ItemProperty HKCU:\\Environment | Select-Object PATH
```

21.2 Registry ändern

Mit `Set-ItemProperty` können Registry-Werte angepasst oder hinzugefügt werden.

```
## Wert ändern
Set-ItemProperty -Path HKCU:\\Environment -Name TestVar -Value "123"

## Wert hinzufügen
New-ItemProperty -Path HKCU:\\Environment -Name NeueVar -Value "abc"
↪  -PropertyType String

## Wert löschen
Remove-ItemProperty -Path HKCU:\\Environment -Name TestVar
```

21.3 Umgebungsvariablen lesen

Umgebungsvariablen sind als Provider verfügbar und lassen sich wie ein HashTable abfragen.

```
## Alle Variablen anzeigen
Get-ChildItem Env:

## Einzelne Variable auslesen
$env:PATH
```

21.4 Umgebungsvariablen ändern

Variablen können direkt gesetzt werden. Diese Änderungen gelten jedoch nur für die aktuelle Session.

```
## Variable setzen
$env:TestVar = "Hallo"

## Variable wieder entfernen
Remove-Item Env:TestVar
```

21.5 Persistente Variablen

Um Variablen dauerhaft zu setzen, müssen sie in der Registry gespeichert oder in Profildateien hinterlegt werden.

```
## Dauerhaft per Registry
Set-ItemProperty -Path HKCU:\\Environment -Name MeineVar -Value
↪  "Persistent"

## In Profil-Skript schreiben
Add-Content -Path $PROFILE -Value "$env:MeineVar = 'Persistent'"
```

21.6 Best Practices

- Änderungen an Registry und Variablen nur mit Bedacht durchführen.
- Immer Backups oder Wiederherstellungspunkte bereithalten.
- Für Tests besser Session-Variablen statt Registry nutzen.
- Änderungen an PATH und kritischen Variablen sorgfältig prüfen.

```powershell
## Vorsichtiger Test
Set-ItemProperty -Path HKCU:\\Environment -Name TestVar -Value "Demo"
↪  -WhatIf
```

22. Prozesse und Dienste

PowerShell bietet Cmdlets, um Prozesse und Dienste zu überwachen und zu steuern. Damit lassen sich Anwendungen starten, beenden oder Services konfigurieren.

```powershell
Get-Help -Name 'about_Processes' -ShowWindow
```

22.1 Prozesse anzeigen

Prozesse können mit `Get-Process` aufgelistet werden. So erhältst du Informationen wie CPU-Auslastung, Speicher oder Prozess-ID.

```powershell
## Alle Prozesse anzeigen
Get-Process

## Spezifischen Prozess anzeigen
Get-Process -Name notepad

## Prozesse sortieren nach CPU
Get-Process | Sort-Object CPU -Descending
```

22.2 Prozesse steuern

Prozesse lassen sich direkt starten oder beenden.

```powershell
## Prozess starten
Start-Process notepad.exe

## Prozess mit Parametern starten
Start-Process notepad.exe -ArgumentList "C:\\Temp\\info.txt"

## Prozess beenden
Stop-Process -Name notepad -Force
```

22.3 Dienste anzeigen

Dienste können mit `Get-Service` überwacht werden. Du kannst Status, Starttyp und Namen abfragen.

```powershell
## Alle Dienste anzeigen
Get-Service

## Dienst nach Namen suchen
Get-Service -Name wuauserv

## Nach Status filtern
Get-Service | Where-Object Status -eq Running
```

22.4 Dienste steuern

Dienste können gestartet, gestoppt oder neu gestartet werden.

```
## Dienst starten
Start-Service -Name wuauserv

## Dienst stoppen
Stop-Service -Name wuauserv

## Dienst neu starten
Restart-Service -Name wuauserv
```

22.5 Dienste konfigurieren

Über WMI oder CIM lassen sich Starttyp und andere Eigenschaften von Diensten ändern.

```
## Starttyp ändern (z. B. auf Automatisch)
Set-Service -Name wuauserv -StartupType Automatic
```

22.6 Best Practices

- Prozesse und Dienste nie blind beenden – mögliche Abhängigkeiten prüfen.
- Administrative Rechte beachten: manche Dienste erfordern erhöhte Privilegien.
- Für wiederkehrende Verwaltung Skripte nutzen, statt manuell zu arbeiten.

```
## Beispiel: Alle gestoppten Dienste starten
Get-Service | Where-Object Status -eq Stopped | Start-Service
```

23. Netzwerk & Verbindungen

PowerShell ermöglicht das Überprüfen von Netzwerkverbindungen, das Abfragen von Schnittstellen und das Testen der Erreichbarkeit von Hosts. Damit lassen sich Fehlerdiagnosen und Netzwerkautomatisierungen durchführen.

```
Get-Help -Name 'Test-Connection' -ShowWindow
```

23.1 Verbindungen testen

Mit `Test-Connection` und `Test-NetConnection` prüfst du die Erreichbarkeit von Systemen und Ports.

```
## Ping auf Host
Test-Connection google.com -Count 2

## Port prüfen
Test-NetConnection -ComputerName google.com -Port 443
```

23.2 IP- und Adapterinformationen

Mit `Get-NetIPAddress` und `Get-NetAdapter` kannst du lokale Netzwerkinformationen abrufen.

```
## IP-Adressen anzeigen
Get-NetIPAddress

## Netzwerkadapter anzeigen
Get-NetAdapter
```

23.3 DNS-Abfragen

DNS-Einträge lassen sich direkt mit PowerShell prüfen.

```
## DNS-Eintrag auflösen
Resolve-DnsName google.com
```

23.4 Offene Ports und Verbindungen

Mit `Get-NetTCPConnection` lassen sich aktuelle TCP-Verbindungen und offene Ports anzeigen.

```
## Alle offenen Verbindungen anzeigen
Get-NetTCPConnection

## Nach RemotePort filtern
Get-NetTCPConnection -RemotePort 443
```

23.5 Dateien aus dem Internet laden

PowerShell kann Dateien direkt von HTTP/HTTPS herunterladen.

```
## Datei herunterladen
Invoke-WebRequest -Uri "https://example.com/Datei.txt" -OutFile
↪  C:\\Temp\\Datei.txt

## Nur Inhalt anzeigen
Invoke-WebRequest -Uri "https://example.com" | Select-Object
↪  -ExpandProperty Content
```

23.6 Best Practices

- Für einfache Pings `Test-Connection`, für Ports und Routing `Test-NetConnection` nutzen.
- Nur benötigte Verbindungen dauerhaft offen lassen.
- Bei automatisierten Downloads immer HTTPS und Zertifikate prüfen.

```
## Verbindung absichern
Invoke-WebRequest -Uri "https://example.com" -SslProtocol Tls12
```

24. Sicherheit & Signaturen

PowerShell bringt Mechanismen mit, um Skripte vor unbefugter Ausführung zu schützen. Dazu gehören Execution Policy, Signaturen und Rechteverwaltung.

```
Get-Help -Name 'about_Signing' -ShowWindow
```

24.1 Execution Policy

Die Execution Policy steuert, welche Skripte ausgeführt werden dürfen. Sie schützt vor unbeabsichtigtem Start unsignierter Skripte.

```
## Aktuelle Execution Policy anzeigen
Get-ExecutionPolicy

## Execution Policy ändern (z. B. RemoteSigned)
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

24.2 Skripte signieren

Mit einem Code-Signing-Zertifikat können Skripte digital signiert werden. Dadurch wird sichergestellt, dass sie unverändert sind und vom Autor stammen.

```
## Zertifikat auswählen
$cert = Get-ChildItem Cert:\CurrentUser\My -CodeSigningCert |
↪  Select-Object -First 1

## Skript signieren
Set-AuthenticodeSignature -FilePath .\MeinSkript.ps1 -Certificate $cert
```

24.3 Signaturen prüfen

Signaturen können jederzeit überprüft werden, um Manipulationen zu erkennen.

```
Get-AuthenticodeSignature -FilePath .\MeinSkript.ps1
```

24.4 Rechte und Rollen

Die Rechteverwaltung in Windows spielt auch in PowerShell eine Rolle. Cmdlets können nur mit den entsprechenden Berechtigungen ausgeführt werden.

```
## Skript als Administrator starten
Start-Process PowerShell -Verb RunAs
```

24.5 Best Practices

- Execution Policy sinnvoll setzen: z. B. **RemoteSigned** für Benutzer, **AllSigned** in Unternehmen.
- Skripte mit vertrauenswürdigen Zertifikaten signieren.
- Nur notwendige Rechte vergeben, Prinzip der minimalen Berechtigung.
- Signaturen regelmäßig prüfen.

```powershell
## Beispiel: Alle Skripte im Ordner prüfen
Get-ChildItem C:\\Scripts\\*.ps1 | Get-AuthenticodeSignature
```

25. PowerShell Profile & Anpassung

Mit Profilskripten lässt sich die PowerShell-Umgebung individuell anpassen. So kannst du eigene Funktionen, Aliase oder Module automatisch laden.

```powershell
Get-Help -Name 'about_Profiles' -ShowWindow
```

25.1 Profilpfade

Jeder Benutzer und jede Host-Anwendung hat ein eigenes Profil. Über die Variable `$PROFILE` erreichst du den aktuellen Pfad.

```powershell
## Pfad zum aktuellen Profil anzeigen
$PROFILE

## Alle Profilpfade anzeigen
$PROFILE | Format-List * -Force
```

25.2 Neues Profil erstellen

Falls noch kein Profil existiert, kannst du eine neue Datei anlegen.

```powershell
## Neues Profil erstellen
if (-not (Test-Path $PROFILE)) {
    New-Item -ItemType File -Path $PROFILE -Force
}
```

25.3 Profil bearbeiten

Das Profil ist eine normale PowerShell-Datei und kann mit einem Editor angepasst werden.

```powershell
## Profil mit VSCode öffnen
code $PROFILE

## Profil mit Notepad öffnen
notepad $PROFILE
```

25.4 Anpassungen im Profil

Typische Anpassungen sind Aliase, Funktionen oder automatische Modulimporte.

```powershell
## Alias hinzufügen
Set-Alias ll Get-ChildItem

## Funktion definieren
function Hallo { "Hallo $env:USERNAME" }

## Modul automatisch laden
```

```
Import-Module PSReadLine
```

25.5 Best Practices

- Profile klar strukturieren und dokumentieren.
- Keine komplexen Skripte direkt im Profil – besser Funktionen oder Module nutzen.
- Änderungen zuerst testen, bevor sie ins Profil übernommen werden.
- Profil regelmäßig sichern und versionskontrolliert ablegen.

```
## Profil sichern
Copy-Item $PROFILE "$env:USERPROFILE\\profile_backup.ps1"
```

26. PSReadLine & Eingabeoptimierung

Das Modul **PSReadLine** verbessert die Eingabe in der PowerShell-Konsole. Es bietet Syntax-Highlighting, Autovervollständigung, Verlaufssuche und viele Anpassungsmöglichkeiten.

```
Get-Help -Name 'about_PSReadLine' -ShowWindow
```

26.1 Syntax-Highlighting

PSReadLine hebt Befehle, Parameter und Strings farblich hervor. Dadurch wird der Code übersichtlicher.

```
## PSReadLine laden (meist standardmäßig aktiv)
Import-Module PSReadLine

## Farben anpassen
Set-PSReadLineOption -Colors @{ "Command" = "Cyan"; "Parameter" = "Yellow"
↪  }
```

26.2 Autovervollständigung

Mit Tab lassen sich Cmdlets, Parameter und Dateien automatisch vervollständigen.

```
## Erweiterte Vervollständigung aktivieren
Set-PSReadLineOption -PredictionSource HistoryAndPlugin
```

26.3 Verlaufssuche

Die Eingabehistorie kann durchsucht und wiederverwendet werden. So sparst du Zeit bei häufig genutzten Befehlen.

```
## Mit STRG+R Rückwärtssuche im Verlauf starten
## Mit Pfeiltasten durch die History navigieren
```

26.4 Tastenkombinationen

Viele Funktionen lassen sich mit Tastenkürzeln aufrufen. Diese können individuell angepasst werden.

```
## Tastenkombination für Clear-Host setzen
Set-PSReadLineKeyHandler -Key Ctrl+l -Function ClearScreen
```

26.5 Best Practices

- Farben und Shortcuts an persönliche Vorlieben anpassen.
- Verlauf regelmäßig bereinigen, falls sensible Daten enthalten sind.
- Autovervollständigung mit Plugins wie **Az.Tools.Predictor** erweitern.

```
## Verlauf löschen
Clear-History
Remove-Item (Get-PSReadLineOption).HistorySavePath
```

27. PowerShell Gallery & Paketmanagement

Die PowerShell Gallery ist das zentrale Repository für Module, Skripte und Ressourcen. Mit Paketmanagement-Cmdlets kannst du Module installieren, aktualisieren und verwalten.

```
Get-Help -Name 'PowerShellGet' -ShowWindow
```

27.1 Verfügbare Repositories anzeigen

Mit `Get-PSRepository` siehst du, welche Repositories eingebunden sind.

```
## Registrierte Repositories anzeigen
Get-PSRepository

## Neues Repository registrieren
Register-PSRepository -Name MeinRepo -SourceLocation
↪  "https://myrepo.com/api/v2"
```

27.2 Module suchen

Module können direkt aus der Gallery gesucht werden.

```
## Modul suchen
Find-Module -Name Pester

## Nach Schlagwort suchen
Find-Module -Tag Testing
```

27.3 Module installieren und aktualisieren

Mit wenigen Befehlen kannst du Module laden oder auf den neuesten Stand bringen.

```
## Modul installieren
Install-Module -Name Pester -Scope CurrentUser

## Modul aktualisieren
Update-Module -Name Pester
```

```
## Modul deinstallieren
Uninstall-Module -Name Pester
```

27.4 Module importieren und verwalten

Installierte Module können manuell geladen oder entfernt werden.

```
## Modul importieren
Import-Module Pester

## Geladene Module anzeigen
Get-Module

## Modul entfernen
Remove-Module Pester
```

27.5 Skripte aus der Gallery

Auch Skripte können direkt aus der Gallery installiert werden.

```
## Skript suchen
Find-Script -Name Start-Build

## Skript installieren
Install-Script -Name Start-Build -Scope CurrentUser
```

27.6 Best Practices

- Module nach Möglichkeit aus vertrauenswürdigen Quellen installieren.
- Repositories klar trennen: **offiziell**, **intern**, **Test**.
- Module regelmäßig aktualisieren, aber vorher testen.
- Abhängigkeiten dokumentieren.

```
## Alle installierten Module und Versionen anzeigen
Get-InstalledModule
```

28. Versionskontrolle mit Git

Git ist das Standardwerkzeug für Versionskontrolle. Mit PowerShell kannst du Git direkt nutzen, um Änderungen nachzuvollziehen, Branches zu verwalten und Code zu teilen.

```
Get-Help -Name 'about_Version_Control' -ShowWindow
```

28.1 Git installieren

Git muss lokal installiert sein, um es aus PowerShell nutzen zu können.

```
## Version prüfen
git --version
```

28.2 Repository erstellen

Ein Git-Repository speichert den Projektverlauf. Es kann lokal oder remote (z. B. GitHub) liegen.

```
## Neues Repository initialisieren
git init

## Repository klonen
git clone https://github.com/benutzer/projekt.git
```

28.3 Änderungen nachverfolgen

Dateien werden in Git versioniert. Änderungen lassen sich jederzeit einsehen.

```
## Status anzeigen
git status

## Änderungen vormerken
git add .

## Commit erstellen
git commit -m "Neue Funktion hinzugefügt"
```

28.4 Branches und Merges

Mit Branches kannst du Funktionen getrennt entwickeln und später zusammenführen.

```
## Branch erstellen
git branch featureX

## In Branch wechseln
git checkout featureX

## Branch zusammenführen
git checkout main
git merge featureX
```

28.5 Remote-Repositories

Um Code zu teilen, werden Repositories auf GitHub, GitLab oder Azure DevOps genutzt.

```
## Remote hinzufügen
git remote add origin https://github.com/benutzer/projekt.git

## Änderungen hochladen
git push origin main

## Änderungen herunterladen
git pull origin main
```

28.6 Best Practices

- Häufig committen mit aussagekräftigen Nachrichten.
- Branches für Features und Bugfixes nutzen.
- Pull Requests für Code-Reviews einsetzen.
- `.gitignore` nutzen, um unnötige Dateien auszuschließen.

```
## Beispiel für .gitignore
*.log
*.tmp
Secrets.json
```

29. Skripte testen mit Pester

Pester ist das Standard-Framework für Tests in PowerShell. Es ermöglicht Unit-Tests, Integrationstests und das Validieren von Skripten und Modulen.

```
Get-Help -Name 'Pester' -ShowWindow
```

29.1 Pester installieren

Pester ist als Modul verfügbar und kann über die PowerShell Gallery installiert werden.

```
## Modul installieren
Install-Module -Name Pester -Scope CurrentUser

## Modul laden
Import-Module Pester
```

29.2 Erstes Testskript

Ein Testskript besteht aus `Describe`-Blöcken (Testsuite) und `It`-Blöcken (Einzeltests).

```
## Datei: Tests.ps1
Describe "Mathematik" {
    It "Addiert Zahlen korrekt" {
        (2 + 3) | Should -Be 5
    }
}
```

```
## Tests ausführen
Invoke-Pester .\Tests.ps1
```

29.3 Funktionen testen

Eigene Funktionen können gezielt getestet werden.

```
## Funktion
function Addiere($a, $b) { $a + $b }

## Test
Describe "Addiere" {
```

```
    It "Addiert korrekt" {
        (Addiere 2 3) | Should -Be 5
    }
}
```

29.4 Mocks verwenden

Mit Mocks kannst du Abhängigkeiten ersetzen und Verhalten simulieren.

```
Describe "Dateiprüfung" {
    Mock Test-Path { $true }
    It "Datei sollte existieren" {
        (Test-Path "C:\\temp\\demo.txt") | Should -Be $true
    }
}
```

29.5 Testberichte erstellen

Pester kann detaillierte Reports erzeugen, die für CI/CD-Pipelines genutzt werden.

```
Invoke-Pester -OutputFormat NUnitXml -OutputFile TestResult.xml
```

29.6 Best Practices

- Tests immer zusammen mit Code entwickeln (Test-Driven Development bevorzugt).
- Kleine, fokussierte Tests schreiben.
- Tests automatisiert in Build-Pipelines einbinden.
- Auch Fehlerszenarien testen.

```
## Beispiel für Fehlertest
{ 1/0 } | Should -Throw
```

30. Automatisierung mit Tasks & Scheduler

PowerShell-Skripte lassen sich über den Windows-Taskplaner oder andere Scheduler automatisch ausführen. Damit kannst du wiederkehrende Aufgaben zuverlässig automatisieren.

```
Get-Help -Name 'ScheduledTasks' -ShowWindow
```

30.1 Task Scheduler manuell

Mit der grafischen Oberfläche des Windows-Taskplaners können PowerShell-Skripte zu bestimmten Zeiten oder Ereignissen gestartet werden.

- Trigger: Zeitplan oder Ereignis
- Aktion: PowerShell.exe mit Skriptpfad
- Bedingungen: z. B. nur bei Netzstrom

30.2 Aufgaben per PowerShell erstellen

Mit den Cmdlets aus dem Modul **ScheduledTasks** lassen sich geplante Aufgaben direkt aus PowerShell anlegen.

```
## Aktion definieren
$action = New-ScheduledTaskAction -Execute "pwsh.exe" -Argument "-File
↪  C:\\Scripts\\Backup.ps1"

## Trigger: täglich um 20 Uhr
$trigger = New-ScheduledTaskTrigger -Daily -At 20:00

## Aufgabe registrieren
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName
↪  "BackupScript" -Description "Backup per PowerShell"
```

30.3 Aufgaben verwalten

Geplante Aufgaben können angezeigt, gestartet oder entfernt werden.

```
## Alle Aufgaben anzeigen
Get-ScheduledTask

## Aufgabe starten
Start-ScheduledTask -TaskName "BackupScript"

## Aufgabe entfernen
Unregister-ScheduledTask -TaskName "BackupScript" -Confirm:$false
```

30.4 Aufgaben mit Rechten ausführen

Standardmäßig laufen Tasks im Benutzerkontext. Sie können aber auch mit höheren Rechten oder unter Systemkonto ausgeführt werden.

```
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName
↪  "BackupAdmin" -User "SYSTEM"
```

30.5 Best Practices

- Aufgaben klar benennen und dokumentieren.
- Skripte mit Logging und Fehlerbehandlung ausstatten.
- Tasks regelmäßig prüfen und aufräumen.
- Nach Möglichkeit **pwsh.exe** statt **powershell.exe** nutzen (PowerShell 7).

```
## Beispiel: Logging im Taskskript
Start-Transcript -Path C:\\Logs\\backup.log
## ... Backup-Code ...
Stop-Transcript
```

31. PowerShell in CI/CD-Pipelines

PowerShell eignet sich hervorragend für Continuous Integration (CI) und Continuous Deployment (CD). Skripte können Build-, Test- und Deployment-Prozesse automatisieren.

```
Get-Help -Name 'PowerShell' -ShowWindow
```

31.1 Einsatz in Build-Systemen

Viele Build-Server wie Azure DevOps, GitHub Actions oder Jenkins unterstützen PowerShell-Skripte nativ.

```
## Beispiel: GitHub Actions Workflow
name: CI
on: [push]
jobs:
  build:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v2
      - name: PowerShell Script
        run: pwsh ./build.ps1
```

31.2 Tests einbinden

Mit Pester lassen sich Unit-Tests automatisch ausführen und Ergebnisse in Reports speichern.

```
## Tests ausführen und Report erzeugen
Invoke-Pester -OutputFormat NUnitXml -OutputFile TestResult.xml
```

31.3 Artefakte erstellen

Skripte können Build-Artefakte erzeugen, wie ZIP-Dateien oder Installationspakete.

```
## Ordner packen
Compress-Archive -Path ./App -DestinationPath ./build/App.zip
```

31.4 Deployment automatisieren

Deployment-Skripte können Dateien kopieren, Dienste neu starten oder Konfigurationen ändern.

```
## Dateien auf Server kopieren
Copy-Item -Path ./build/App.zip -Destination \\Server01\Deploy$

## Dienst neu starten
Restart-Service -Name IIS
```

31.5 Best Practices

- Skripte modular aufbauen, damit sie in verschiedenen Pipelines nutzbar sind.
- Tests automatisiert in die Pipeline integrieren.
- Fehlerbehandlung einbauen, damit Pipelines bei Problemen korrekt stoppen.
- Konfigurationen und Secrets niemals fest im Skript speichern, sondern über Umgebungsvariablen oder Secret Stores.

```powershell
## Beispiel für Zugriff auf Secret in Pipeline
$apiKey = $env:API_KEY
```

32. PowerShell und REST-APIs

Mit PowerShell lassen sich REST-APIs direkt ansprechen. Über `Invoke-RestMethod` und `Invoke-WebRequest` können Daten abgerufen, erstellt oder geändert werden.

```powershell
Get-Help -Name 'Invoke-RestMethod' -ShowWindow
```

32.1 Daten von APIs abrufen

Mit `Invoke-RestMethod` erhältst du JSON oder XML direkt als Objekte, die weiterverarbeitet werden können.

```powershell
## JSON von API abrufen
Invoke-RestMethod -Uri
↪  "https://api.github.com/repos/PowerShell/PowerShell"

## XML von API abrufen
Invoke-RestMethod -Uri "https://www.w3schools.com/xml/note.xml"
```

32.2 API mit Parametern aufrufen

Viele APIs erwarten Parameter, die in der URL oder im Body übergeben werden.

```powershell
## Mit Query-Parametern
Invoke-RestMethod -Uri "https://api.agify.io/?name=attila"
```

32.3 Authentifizierung

REST-APIs erfordern oft Token oder Anmeldedaten. Diese können im Header mitgegeben werden.

```powershell
## Mit API-Key im Header
$headers = @{ Authorization = "Bearer <TOKEN>" }
Invoke-RestMethod -Uri "https://api.example.com/data" -Headers $headers
```

32.4 Daten an API senden

Mit POST, PUT oder DELETE lassen sich Daten ändern oder neue Einträge erstellen.

```powershell
## POST-Daten senden
$body = @{ Name = "Test"; Wert = 42 } | ConvertTo-Json
Invoke-RestMethod -Uri "https://api.example.com/items" -Method Post -Body
↪  $body -ContentType "application/json"
```

32.5 Fehlerbehandlung

APIs liefern häufig Statuscodes zurück, die beachtet werden sollten.

```powershell
try {
    Invoke-RestMethod -Uri "https://api.example.com/invalid"
}
catch {
    $_.Exception.Response.StatusCode.Value__
}
```

32.6 Best Practices

- Immer HTTPS verwenden, niemals unsichere HTTP-Verbindungen.
- API-Keys und Tokens sicher speichern (z. B. SecretStore, Azure Key Vault).
- Ergebnisse in Objekte umwandeln, um sie direkt weiterzuverarbeiten.
- Rate-Limits und Quotas von APIs beachten.

```powershell
## Ergebnis in Datei sichern
Invoke-RestMethod -Uri
↪  "https://api.github.com/repos/PowerShell/PowerShell/releases" | \
  ConvertTo-Json | Out-File C:\\Temp\\releases.json
```

33. JSON, XML & CSV verarbeiten

PowerShell unterstützt den Umgang mit strukturierten Datenformaten wie JSON, XML und CSV nativ. So lassen sich Konfigurationsdateien, API-Daten oder Reports einfach einlesen und weiterverarbeiten.

```powershell
Get-Help -Name 'ConvertFrom-Json' -ShowWindow
```

33.1 JSON verarbeiten

JSON wird direkt in PowerShell-Objekte umgewandelt und kann einfach genutzt werden.

```powershell
## JSON einlesen
$json = '{"Name":"Attila","Alter":38}' | ConvertFrom-Json
$json.Name

## Objekt in JSON konvertieren
$obj = @{ Projekt = "Demo"; Status = "Aktiv" }
$obj | ConvertTo-Json -Depth 3
```

33.2 XML verarbeiten

XML wird als XML-Objekt eingelesen und lässt sich mit Pfaden und Eigenschaften durchsuchen.

```powershell
## XML einlesen
[xml]$xml = Get-Content C:\\Temp\\Daten.xml
$xml.Dokument.Element

## Neues XML erzeugen
$xml = New-Object System.Xml.XmlDocument
```

```
$root = $xml.CreateElement("Root")
$xml.AppendChild($root)
$xml.Save("C:\\Temp\\neu.xml")
```

33.3 CSV verarbeiten

CSV-Dateien werden automatisch in Tabellenobjekte umgewandelt. Ideal für Reports oder Datenexporte.

```
## CSV einlesen
$data = Import-Csv C:\\Temp\\Personen.csv
$data | Where-Object { $_.Alter -gt 30 }

## CSV exportieren
$data | Export-Csv C:\\Temp\\export.csv -NoTypeInformation -Encoding UTF8
```

33.4 Konvertierungen

PowerShell kann zwischen den Formaten wechseln, um Daten flexibel zu nutzen.

```
## JSON nach CSV konvertieren
Invoke-RestMethod -Uri "https://api.agify.io/?name=attila" | \
  ConvertTo-Csv -NoTypeInformation

## CSV nach JSON konvertieren
Import-Csv C:\\Temp\\Personen.csv | ConvertTo-Json -Depth 3
```

33.5 Best Practices

- Für APIs JSON bevorzugen, für Tabellenberichte CSV.
- Bei JSON `-Depth` beachten, da verschachtelte Strukturen sonst abgeschnitten werden.
- CSV-Export immer mit `-NoTypeInformation` nutzen.
- XML nur verwenden, wenn zwingend notwendig (z. B. für Legacy-Systeme).

```
## Beispiel: JSON-Ergebnis sichern
Invoke-RestMethod -Uri
↪  "https://api.github.com/repos/PowerShell/PowerShell" | \
  ConvertTo-Json -Depth 5 | Out-File C:\\Temp\\repo.json
```

34. PowerShell und WMI/CIM

WMI (Windows Management Instrumentation) und CIM (Common Information Model) ermöglichen den Zugriff auf Systeminformationen und Verwaltungseinstellungen. PowerShell bietet Cmdlets, um diese Schnittstellen direkt zu nutzen.

```
Get-Help -Name 'about_WMI' -ShowWindow
```

34.1 WMI vs. CIM

- **WMI**: Ältere Technologie, lokal und per DCOM nutzbar.

- **CIM**: Neuer Standard, basiert auf WS-Man (WinRM), plattformübergreifend.

Empfehlung: Für neue Skripte **CIM** nutzen.

34.2 Informationen abfragen

Mit `Get-CimInstance` kannst du Systeminformationen einfach abrufen.

```
## Betriebssystem-Infos
Get-CimInstance Win32_OperatingSystem | Select-Object Caption, Version,
↪  LastBootUpTime

## Prozesse anzeigen
Get-CimInstance Win32_Process | Select-Object Name, ProcessId

## Services abfragen
Get-CimInstance Win32_Service | Where-Object { $_.State -eq "Running" }
```

34.3 Remotezugriff

CIM arbeitet über WS-Man und ist daher für Remoting optimiert.

```
## Remote-Computer abfragen
Get-CimInstance Win32_ComputerSystem -ComputerName Server01

## Mit CIM-Session
$session = New-CimSession -ComputerName Server01
Get-CimInstance Win32_OperatingSystem -CimSession $session
Remove-CimSession $session
```

34.4 Aktionen durchführen

Neben Abfragen lassen sich auch Aktionen über WMI/CIM starten.

```
## Prozess starten
Invoke-CimMethod -ClassName Win32_Process -MethodName Create -Arguments @{
↪  CommandLine = "notepad.exe" }

## Dienst stoppen
Invoke-CimMethod -ClassName Win32_Service -MethodName StopService -Key @{
↪  Name = "wuauserv" }
```

34.5 Unterschiede zu Get-WmiObject

`Get-WmiObject` ist veraltet und sollte nicht mehr verwendet werden. Stattdessen `Get-CimInstance` nutzen.

```
## Altes Cmdlet (vermeiden)
Get-WmiObject Win32_OperatingSystem

## Neues Cmdlet
Get-CimInstance Win32_OperatingSystem
```

34.6 Best Practices

- Für Skripte **CIM-Cmdlets** bevorzugen.
- Für wiederholte Abfragen CIM-Sessions verwenden.
- Nur benötigte Eigenschaften abrufen, um Performance zu verbessern.
- Auf Remotesystemen WinRM aktivieren, falls CIM genutzt wird.

```
## Beispiel: Nur bestimmte Eigenschaften abrufen
Get-CimInstance Win32_NetworkAdapterConfiguration | Select-Object
↪  Description, MACAddress, IPEnabled
```

35. Active Directory Verwaltung

Mit PowerShell und dem Modul **ActiveDirectory** lassen sich Benutzer, Gruppen, Computer und OU-Strukturen zentral verwalten. Das Modul ist Teil der RSAT-Tools oder auf Domain Controllern bereits installiert.

```
Get-Help -Name ActiveDirectory -ShowWindow
```

35.1 Modul laden

Bevor du Active Directory-Cmdlets nutzen kannst, muss das Modul geladen werden.

```
## Modul importieren
Import-Module ActiveDirectory

## Verfügbare Cmdlets anzeigen
Get-Command -Module ActiveDirectory
```

35.2 Benutzer verwalten

Benutzer können erstellt, geändert oder gelöscht werden.

```
## Neuen Benutzer anlegen
New-ADUser -Name "Max Mustermann" -SamAccountName mmustermann
↪  -AccountPassword (Read-Host -AsSecureString "Passwort") -Enabled $true

## Benutzer ändern
Set-ADUser -Identity mmustermann -Department "IT"

## Benutzer anzeigen
Get-ADUser -Identity mmustermann -Properties *

## Benutzer löschen
Remove-ADUser -Identity mmustermann -Confirm:$false
```

35.3 Gruppen verwalten

Gruppen steuern Berechtigungen und lassen sich per Cmdlets anpassen.

```
## Neue Gruppe erstellen
New-ADGroup -Name "IT-Admins" -GroupScope Global -Path
↪  "OU=Gruppen,DC=contoso,DC=com"
```

```
## Benutzer zur Gruppe hinzufügen
Add-ADGroupMember -Identity "IT-Admins" -Members mmustermann

## Gruppenmitglieder anzeigen
Get-ADGroupMember -Identity "IT-Admins"

## Benutzer aus Gruppe entfernen
Remove-ADGroupMember -Identity "IT-Admins" -Members mmustermann
↪  -Confirm:$false
```

35.4 Computer und OUs

Auch Computerobjekte und Organisationseinheiten lassen sich verwalten.

```
## Computerobjekt anzeigen
Get-ADComputer -Identity PC01 -Properties *

## Computer verschieben
Move-ADObject -Identity "CN=PC01,OU=Workstations,DC=contoso,DC=com"
↪  -TargetPath "OU=IT,DC=contoso,DC=com"

## Neue OU erstellen
New-ADOrganizationalUnit -Name "Skripte" -Path "DC=contoso,DC=com"
```

35.5 Suchen und filtern

AD-Objekte lassen sich flexibel filtern und suchen.

```
## Alle Benutzer in der OU "IT"
Get-ADUser -Filter * -SearchBase "OU=IT,DC=contoso,DC=com"

## Benutzer mit abgelaufenem Passwort
Search-ADAccount -PasswordExpired

## Gesperrte Konten anzeigen
Search-ADAccount -LockedOut
```

35.6 Best Practices

- Änderungen zuerst in Testumgebungen durchführen.
- Immer mit `-WhatIf` prüfen, bevor produktive Objekte geändert werden.
- OU-Struktur sauber halten und Namenskonventionen einhalten.
- Gruppenverschachtelungen vermeiden, um Berechtigungen übersichtlich zu halten.

```
## Sicher testen
Remove-ADUser -Identity testuser -WhatIf
```

36. Exchange & Office 365 Verwaltung

Mit PowerShell lassen sich sowohl lokale Exchange-Server als auch Exchange Online (Office 365) administrieren. Dazu stehen eigene Module und Cmdlets bereit.

```
Get-Help -Name Exchange -ShowWindow
```

36.1 Exchange-Modul laden

Für lokale Exchange-Server wird das Exchange Management Shell-Modul verwendet. Für Exchange Online ist das Modul **ExchangeOnlineManagement** nötig.

```
## Exchange Online Modul installieren
Install-Module -Name ExchangeOnlineManagement -Scope CurrentUser

## Modul importieren
Import-Module ExchangeOnlineManagement
```

36.2 Verbindung zu Exchange Online

Zur Verwaltung von Exchange Online wird eine Authentifizierung benötigt.

```
## Verbindung herstellen
Connect-ExchangeOnline -UserPrincipalName admin@contoso.com

## Verbindung trennen
Disconnect-ExchangeOnline
```

36.3 Postfächer verwalten

Postfächer lassen sich erstellen, ändern und anzeigen.

```
## Alle Postfächer anzeigen
Get-Mailbox

## Neues Postfach anlegen (lokal)
New-Mailbox -Name "Max Mustermann" -UserPrincipalName
↪  mmustermann@contoso.com -Password (Read-Host -AsSecureString
↪  "Passwort")

## Eigenschaften ändern
Set-Mailbox -Identity mmustermann -EmailAddresses
↪  @{Add="max.mustermann@contoso.com"}
```

36.4 Verteiler und Gruppen

Verteilerlisten und Gruppen können per Cmdlet erstellt und verwaltet werden.

```
## Neue Verteilergruppe
New-DistributionGroup -Name "IT-Team" -PrimarySmtpAddress it@contoso.com

## Mitglieder hinzufügen
```

```
Add-DistributionGroupMember -Identity "IT-Team" -Member mmustermann

## Mitglieder anzeigen
Get-DistributionGroupMember -Identity "IT-Team"
```

36.5 Exchange-Richtlinien

Mit Richtlinien können z. B. Mailbox Limits oder Archivierung konfiguriert werden.

```
## Mailboxgröße prüfen
Get-MailboxStatistics -Identity mmustermann

## Archiv aktivieren
Enable-Mailbox -Identity mmustermann -Archive
```

36.6 Best Practices

- In Office 365 immer moderne Authentifizierung verwenden.
- Cmdlets regelmäßig in der Doku prüfen, da sich Online-Module schnell ändern.
- Skripte für Massenänderungen nutzen, statt GUI.
- Nach größeren Änderungen Postfach-Statistiken prüfen.

```
## Beispiel: Alle Postfächer mit Archiv
Get-Mailbox | Where-Object { $_.ArchiveStatus -eq "Active" }
```

37. Windows Management (Updates, Eventlogs, Tasks)

PowerShell eignet sich, um Windows-Systeme umfassend zu verwalten: Updates installieren, Ereignisprotokolle auswerten und geplante Aufgaben steuern.

```
Get-Help -Name 'about_Eventlogs' -ShowWindow
```

37.1 Windows Updates

Mit Modulen wie **PSWindowsUpdate** lassen sich Updates per PowerShell installieren und verwalten.

```
## Modul installieren
Install-Module -Name PSWindowsUpdate -Scope CurrentUser

## Verfügbare Updates anzeigen
Get-WindowsUpdate

## Updates installieren
Install-WindowsUpdate -AcceptAll -AutoReboot
```

37.2 Ereignisprotokolle

Ereignisprotokolle helfen bei Fehleranalyse und Monitoring. PowerShell bietet verschiedene Cmdlets zum Auslesen.

```
## Alle Logs anzeigen
Get-EventLog -List

## Einträge aus dem System-Log
Get-EventLog -LogName System -Newest 20

## Neue API (empfohlen)
Get-WinEvent -LogName Application -MaxEvents 10

## Nach Fehlern filtern
Get-WinEvent -FilterHashtable @{LogName='System'; Level=2} | Format-Table
↪  TimeCreated, Message
```

37.3 Geplante Aufgaben

Windows Scheduled Tasks können direkt mit PowerShell verwaltet werden.

```
## Aufgaben anzeigen
Get-ScheduledTask

## Aufgabe starten
Start-ScheduledTask -TaskName "BackupScript"

## Aufgabe anlegen
$action = New-ScheduledTaskAction -Execute "pwsh.exe" -Argument "-File
↪  C:\\Scripts\\Backup.ps1"
$trigger = New-ScheduledTaskTrigger -Daily -At 20:00
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName
↪  "BackupScript"
```

37.4 Dienste und Systemstatus

Systemdienste und Statuswerte können zentral überwacht werden.

```
## Alle Dienste anzeigen
Get-Service

## Bestimmten Dienst neu starten
Restart-Service -Name wuauserv

## Computerinformationen
Get-ComputerInfo | Select-Object OSName, OsArchitecture, WindowsVersion
```

37.5 Best Practices

- Für Updates nur signierte Quellen nutzen.
- Ereignisprotokolle gezielt filtern, statt alles zu exportieren.
- Geplante Aufgaben dokumentieren und mit klaren Namen versehen.
- Services nicht blind stoppen – Abhängigkeiten prüfen.

```
## Beispiel: Nur Fehlereinträge der letzten 24h
Get-WinEvent -FilterHashtable @{LogName='Application'; Level=2;
↪  StartTime=(Get-Date).AddDays(-1)}
```

38. JEA (Just Enough Administration)

Just Enough Administration (JEA) ist ein Sicherheitskonzept in PowerShell, das es ermöglicht, minimale Rechte für Verwaltungsaufgaben zu vergeben. Nutzer erhalten nur die Befehle, die sie für ihre Arbeit benötigen.

```
Get-Help -Name 'about_JEA' -ShowWindow
```

38.1 Grundlagen

Mit JEA kannst du sogenannte Endpoints definieren, über die Benutzer eingeschränkte PowerShell-Sitzungen starten. Diese Sitzungen enthalten nur die erlaubten Cmdlets, Funktionen oder Skripte.

- Prinzip: **Least Privilege** – nur die notwendigen Rechte
- Vorteile: Weniger Angriffsfläche, kontrollierte Administration

38.2 Role Capabilities

Role Capabilities definieren, welche Cmdlets oder Funktionen ein Benutzer in einer JEA-Session ausführen darf.

```
## Beispiel: Role Capability Datei erstellen
New-Item -Path "C:\\Program
↪  Files\\WindowsPowerShell\\Modules\\MyJEA\\RoleCapabilities" -ItemType
↪  Directory -Force

$rcFile = "C:\\Program
↪  Files\\WindowsPowerShell\\Modules\\MyJEA\\RoleCapabilities\\HelpDesk.psrc"

New-PSRoleCapabilityFile -Path $rcFile -VisibleCmdlets Get-Service,
↪  Restart-Service
```

38.3 Session Configuration

Eine Session Configuration bindet die Role Capabilities an eine PowerShell-Endpunktdefinition.

```
## Session Configuration Datei erstellen
$sessionFile = "C:\\Program
↪  Files\\WindowsPowerShell\\Modules\\MyJEA\\MyJEA.pssc"

New-PSSessionConfigurationFile -Path $sessionFile -SessionType
↪  RestrictedRemoteServer -RoleDefinitions @{ "CONTOSO\\HelpDesk" = @{
↪  RoleCapabilities = 'HelpDesk' } }

## Registrierung der Session
```

```
Register-PSSessionConfiguration -Name "JEA-HelpDesk" -Path $sessionFile
↪  -Force
```

38.4 Nutzung von JEA-Sessions

Benutzer können sich nun mit dem JEA-Endpunkt verbinden und haben nur die erlaubten Befehle zur Verfügung.

```
## Verbindung mit JEA-Endpoint
Enter-PSSession -ComputerName Server01 -ConfigurationName JEA-HelpDesk
```

38.5 Best Practices

- JEA für Rollen wie Helpdesk, Support oder Operator nutzen.
- Nur die nötigsten Cmdlets und Funktionen freigeben.
- Session Configurations versionieren und dokumentieren.
- JEA-Endpunkte regelmäßig überprüfen und anpassen.

```
## Verfügbare JEA-Endpoints anzeigen
Get-PSSessionConfiguration
```

39. Linux & Cross-Plattform PowerShell

Seit PowerShell 7 ist PowerShell plattformübergreifend verfügbar und läuft auf Windows, Linux und macOS. Dadurch können Skripte für heterogene Umgebungen entwickelt werden.

```
Get-Help -Name 'about_CrossPlatform' -ShowWindow
```

39.1 Installation auf Linux

PowerShell kann über Paketmanager wie `apt`, `yum` oder `zypper` installiert werden.

```
## Ubuntu/Debian
sudo apt-get update
sudo apt-get install -y powershell

## CentOS/RHEL
sudo yum install -y powershell

## Starten
pwsh
```

39.2 Unterschiede zu Windows

- Standard-Shell ist Bash, daher andere Tools verfügbar
- Pfade: `/home/user` statt `C:\\Users\\user`
- Dienste über `systemctl` statt `Get-Service`

```
## Beispiel: Prozessabfrage auf Linux
Get-Process --Name sshd
```

```
## Dateioperationen sind gleich
Get-ChildItem /var/log
```

39.3 Plattformübergreifende Skripte

Skripte sollten so geschrieben werden, dass sie auf allen Plattformen lauffähig sind.

```
## Betriebssystem prüfen
if ($IsWindows) { "Windows" }
elseif ($IsLinux) { "Linux" }
elseif ($IsMacOS) { "macOS" }
```

39.4 SSH-Remoting

Auf Linux erfolgt Remoting über SSH statt WinRM. Dies ermöglicht sichere plattformübergreifende Administration.

```
## Verbindung per SSH
Enter-PSSession -HostName server01.linux.local -UserName admin
```

39.5 Best Practices

- Plattformunterschiede in Skripten berücksichtigen.
- Nur Cmdlets verwenden, die auf allen Plattformen funktionieren.
- Für spezifische Plattformbefehle Bedingungen einbauen (`$IsWindows`, `$IsLinux`).
- Wo möglich Standards wie SSH, JSON und REST nutzen.

```
## Beispiel: plattformabhängige Logdateien
if ($IsWindows) { Get-Content C:\\Windows\\WindowsUpdate.log }
else { Get-Content /var/log/syslog }
```

40. PowerShell und .NET-Integration

PowerShell basiert auf .NET und kann direkt auf Klassen, Methoden und Bibliotheken zugreifen. Damit lassen sich Funktionen nutzen, die über die Standard-Cmdlets hinausgehen.

```
Get-Help -Name 'about_Classes' -ShowWindow
```

40.1 .NET-Klassen verwenden

Du kannst .NET-Klassen direkt instanziieren und deren Methoden aufrufen.

```
## String-Objekt erstellen
$str = New-Object System.Text.StringBuilder
$str.Append("Hallo ")
$str.Append("Welt")
$str.ToString()
```

40.2 Statische Methoden und Eigenschaften

Statische Mitglieder lassen sich ohne Objekt direkt über den Klassennamen aufrufen.

```powershell
## Zufallszahl
[System.Random]::new().Next(1,100)

## Aktuelles Datum
[System.DateTime]::Now
```

40.3 Dateien und Streams

Mit .NET lassen sich auch komplexere Dateioperationen durchführen.

```powershell
## Datei schreiben
[System.IO.File]::WriteAllText("C:\\Temp\\test.txt", "Hallo Welt")

## Datei lesen
[System.IO.File]::ReadAllText("C:\\Temp\\test.txt")
```

40.4 Assemblies laden

Externe .NET-Bibliotheken können in PowerShell eingebunden werden.

```powershell
## DLL laden
Add-Type -Path "C:\\Libs\\MeineLib.dll"

## Klasse aus Assembly verwenden
[MeineLib.Tools]::DoSomething()
```

40.5 Eigene Klassen in PowerShell

Ab PowerShell 5 können eigene Klassen direkt im Skript definiert werden.

```powershell
class Person {
    [string]$Name
    [int]$Alter

    Person([string]$n, [int]$a) {
        $this.Name = $n
        $this.Alter = $a
    }

    [string] Begruessen() {
        return "Hallo, mein Name ist $($this.Name) und ich bin
        ↪  $($this.Alter) Jahre alt."
    }
}

$p = [Person]::new("Attila", 38)
$p.Begruessen()
```

40.6 Best Practices

- .NET nur dort einsetzen, wo Cmdlets nicht ausreichen.

- Auf Kompatibilität achten: Nicht alle .NET-APIs sind auf allen Plattformen verfügbar.
- Für wiederkehrende Logik eigene Klassen und Methoden nutzen.
- Assemblies versionieren und sauber dokumentieren.

```
## Beispiel: GUID generieren
[System.Guid]::NewGuid()
```

41. GUI-Tools mit PowerShell

PowerShell kann nicht nur in der Konsole genutzt werden, sondern auch grafische Oberflächen erzeugen. Dazu lassen sich Windows Forms (WinForms) oder Windows Presentation Foundation (WPF) einsetzen.

```
Get-Help -Name 'Show-Command' -ShowWindow
```

41.1 Grundlagen

- **WinForms**: Einfach, schnell für kleine Tools.
- **WPF**: Moderner, flexibler, trennt Oberfläche (XAML) und Logik.

PowerShell kann beide Varianten direkt verwenden, da sie auf .NET basieren.

41.2 Einfache WinForms

Mit WinForms lassen sich schnell kleine Dialoge bauen.

```
Add-Type -AssemblyName System.Windows.Forms

$form = New-Object Windows.Forms.Form
$form.Text = "Demo-Formular"
$form.Size = '300,200'

$button = New-Object Windows.Forms.Button
$button.Text = "Klick mich"
$button.Dock = 'Fill'
$button.Add_Click({ [Windows.Forms.MessageBox]::Show("Hallo Welt") })

$form.Controls.Add($button)
$form.ShowDialog()
```

41.3 WPF mit XAML

Mit WPF lassen sich komplexere GUIs über XAML-Dateien beschreiben.

```
Add-Type -AssemblyName PresentationFramework

[xml]$xaml = @"
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="Demo" Height="200" Width="300">
    <Grid>
        <Button Name="BtnHallo" Content="Sag Hallo"
↪  HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

```
    </Grid>
</Window>
"@

$reader = (New-Object System.Xml.XmlNodeReader $xaml)
$window = [Windows.Markup.XamlReader]::Load($reader)

$button = $window.FindName("BtnHallo")
$button.Add_Click({ [System.Windows.MessageBox]::Show("Hallo aus WPF") })

$window.ShowDialog()
```

41.4 Events und Logik

GUI-Elemente können Events auslösen, die mit Skriptlogik verknüpft werden.

```
$button.Add_Click({
    $form.Text = "Button wurde geklickt!"
})
```

41.5 Best Practices

- Für kleine Tools → WinForms nutzen.
- Für komplexere Anwendungen → WPF mit XAML.
- Logik und Oberfläche trennen (Code vs. XAML).
- GUI-Tools dokumentieren und testen – sie sind fehleranfälliger als reine Skripte.

```
## Beispiel: Einfaches Eingabefeld
Add-Type -AssemblyName Microsoft.VisualBasic
$eingabe = [Microsoft.VisualBasic.Interaction]::InputBox("Bitte Namen
↪  eingeben:", "Eingabe", "Attila")
"Hallo $eingabe"
```

42. Best Practices & Standards

Eine saubere Arbeitsweise mit PowerShell erleichtert Wartung, Lesbarkeit und Sicherheit von Skripten. Die folgenden Standards helfen bei der täglichen Praxis.

42.1 Namenskonventionen

- Cmdlets im **Verb-Noun**-Format (z. B. `Get-Report`, `Set-Config`).
- Funktions- und Variablennamen klar und sprechend wählen.
- Nur englische Typnamen im Code (z. B. `[string]`, `[int]`).

```
function Get-UserReport {
    param([string]$UserName)
    "Report für $UserName"
}
```

42.2 Kommentare & Dokumentation

- Jede Funktion kurz beschreiben.
- Parameter mit `[Parameter()]` und Attributen dokumentieren.
- Bei komplexeren Skripten Header-Kommentare verwenden.

```
<#!
.SYNOPSIS
    Erstellt einen User-Report
.DESCRIPTION
    Diese Funktion erstellt eine einfache Übersicht für einen Benutzer.
##>
function Get-UserReport { ... }
```

42.3 Fehlerbehandlung

- Immer terminierende Fehler mit `-ErrorAction Stop` erzwingen.
- `try/catch` nur einsetzen, wenn sinnvoll behandelt werden kann.
- Eigene Fehler mit `throw` oder `Write-Error` erzeugen.

```
try {
    Get-Item "C:\\Temp\\config.json" -ErrorAction Stop
}
catch {
    Write-Error "Konfigurationsdatei fehlt!"
}
```

42.4 Sicherheit

- Keine Passwörter im Klartext speichern.
- Für Secrets den **SecretManagement**-Modul oder Credential Manager nutzen.
- Execution Policy sinnvoll setzen (`RemoteSigned` oder `AllSigned`).

```
## Secret sicher ablegen
$cred = Get-Credential
Set-Secret -Name ApiUser -Secret $cred
```

42.5 Performance

- Nur benötigte Eigenschaften abfragen (`Select-Object`).
- Bei großen Datenmengen Pipelines nutzen.
- Wo möglich parallele Ausführung (Jobs, ForEach-Object -Parallel).

```
## Nur benötigte Felder abrufen
Get-Process | Select-Object Name, CPU
```

42.6 Code-Qualität

- Einheitliche Einrückung (4 Leerzeichen).
- Linter wie **PSScriptAnalyzer** verwenden.
- Skripte mit Git versionieren.

```powershell
## Skriptanalyse starten
Invoke-ScriptAnalyzer -Path .\MeinSkript.ps1
```

42.7 Best Practices zusammengefasst

- Klares Cmdlet-Schema (Verb-Noun) nutzen.
- Fehlerbehandlung bewusst einsetzen.
- Security by Default – keine Klartext-Passwörter.
- Performance im Blick behalten.
- Einheitliche Code-Standards und Versionskontrolle einhalten.