

Module

Irgendwann wird jedes Skript groß genug, um unübersichtlich zu werden. Module helfen dabei, Funktionen, Variablen und andere Bestandteile sinnvoll zu organisieren und wiederzuverwenden. Kurz gesagt: weniger Chaos, mehr Struktur. Meistens jedenfalls.

Modules.psm1 – Erweiterungen für das Skript, die zusätzliche Variablen , Funktionen oder Objekte beinhalten. Es kann nicht allein ausgeführt werden und muss von einem **Skript** oder [Manifest importiert](#) werden .

Es gibt auch → **modulqualifizierter Aufruf**

- [Get-Module](#)
- [Import-Module](#)
- [Manifest \(.psd1\)](#)

Get-Module

`Get-Module` listet PowerShell-Module auf. Standardmäßig werden **nur bereits geladene Module** angezeigt.

Mit zusätzlichen Parametern lassen sich auch **installierte, aber noch nicht geladene Module** oder bestimmte Module anhand ihres Namens ermitteln.

Ein Modul stellt Funktionen, Cmdlets, Variablen, Aliase und weitere Inhalte bereit, die PowerShell erweitern.

Grundlagen

Bereits geladene Module anzeigen

```
Get-Module
```

Ausgabe (Beispiel)

```
ModuleType Version Name
-----
Manifest    7.0.0.0 Microsoft.PowerShell.Management
Manifest    7.0.0.0 Microsoft.PowerShell.Utility
Script      2.2.5   PSReadLine
```

Alle installierten Module anzeigen

```
Get-Module -ListAvailable
```

Hierbei werden auch Module angezeigt, die aktuell **nicht geladen** sind.

Ein bestimmtes Modul anzeigen

```
Get-Module Microsoft.PowerShell.Utility
```

oder

```
Get-Module -Name Microsoft.PowerShell.Utility
```

Wildcards verwenden

```
Get-Module Microsoft.*
```

Parameter

Parameter	Beschreibung
-Name	Gibt nur Module mit dem angegebenen Namen zurück. Wildcards sind möglich.
-ListAvailable	Zeigt alle installierten Module an.
-All	Zeigt alle Versionen eines geladenen Moduls an.
-FullyQualifiedName	Sucht anhand eines Modulnamens und optional einer Version oder GUID.
-Refresh	Aktualisiert Informationen über dynamische Module.
-PSEdition	Filtert nach der unterstützten PowerShell-Edition (Desktop oder Core).

Eigenschaften

Die von `Get-Module` zurückgegebenen Objekte besitzen unter anderem folgende Eigenschaften.

Eigenschaft	Beschreibung
Name	Name des Moduls
Version	Versionsnummer
ModuleType	Typ des Moduls
Path	Speicherort der Moduldatei
Guid	Eindeutige Modul-ID

Eigenschaft	Beschreibung
Description	Beschreibung des Moduls
Author	Autor des Moduls
CompanyName	Herausgeber
PowerShellVersion	Erforderliche PowerShell-Version
ClrVersion	Erforderliche .NET-CLR-Version
ExportedFunctions	Exportierte Funktionen
ExportedCmdlets	Exportierte Cmdlets
ExportedAliases	Exportierte Aliase
ExportedVariables	Exportierte Variablen

Beispiele

Geladene Module anzeigen

```
Get-Module
```

Alle installierten Module anzeigen

```
Get-Module -ListAvailable
```

Nur Module eines Herstellers anzeigen

```
Get-Module Microsoft.* -ListAvailable
```

Informationen zu einem Modul abrufen

```
Get-Module PSReadLine
```

Alle installierten Versionen anzeigen

```
Get-Module PSReadLine -ListAvailable -All
```

Exportierte Funktionen eines Moduls anzeigen

```
(Get-Module PSReadLine).ExportedFunctions.Keys
```

Exportierte Cmdlets anzeigen

```
(Get-Module Microsoft.PowerShell.Management).ExportedCmdlets.Keys
```

Installationspfad eines Moduls anzeigen

```
(Get-Module PSReadLine).Path
```

ModuleType

Die Eigenschaft `ModuleType` beschreibt, aus welcher Art von Modul die Informationen stammen.

Typ	Beschreibung
<code>Manifest</code>	Modul basiert auf einer Manifestdatei (<code>.psd1</code>).
<code>Script</code>	PowerShell-Skriptmodul (<code>.psm1</code>).
<code>Binary</code>	Kompiliertes .NET-Modul (<code>.dll</code>).
<code>Dynamic</code>	Zur Laufzeit erzeugtes Modul.
<code>Cim</code>	CIM-Modul (selten verwendet).

Hinweise

- Ohne Parameter werden **nur geladene Module** angezeigt.
- `-ListAvailable` durchsucht die Verzeichnisse aus `$env:PSModulePath`.
- Ein installiertes Modul erscheint erst ohne `-ListAvailable`, nachdem es mit `Import-Module` geladen wurde.
- Ein Modul kann in mehreren Versionen gleichzeitig installiert sein.
- `Get-Module` liefert Objekte vom Typ `System.Management.Automation.PSModuleInfo` zurück.

Verwandte Befehle

Befehl	Beschreibung
<code>Import-Module</code>	Lädt ein Modul in die aktuelle Sitzung.
<code>Remove-Module</code>	Entfernt ein geladenes Modul aus der Sitzung.
<code>Get-Command</code>	Zeigt die Befehle eines Moduls an.
<code>Get-InstalledModule</code>	Zeigt mit PowerShellGet installierte Module an.
<code>Find-Module</code>	Sucht Module in einem Repository (z. B. PSGallery).

Typische Kombinationen

```
# Alle installierten Module
```

```
Get-Module -ListAvailable
```

```
# Alle Befehle eines Moduls
```

```
Get-Command -Module Microsoft.PowerShell.Management
```

```
# Modul laden
```

```
Import-Module PSReadLine
```

```
# Danach prüfen, ob es geladen wurde
```

```
Get-Module PSReadLine
```

Import-Module

Hier passiert das Offensichtliche, das trotzdem erstaunlich oft falsch gemacht wird: Du lädst dein Modul in die aktuelle Session.

Ohne Import bist du einfach nur jemand, der Code hortet.

□□ Grundbefehl

```
Import-Module ModulName
```

PowerShell sucht dein Modul automatisch in den bekannten Modulpfaden (`$env:PSModulePath`).

Dabei reicht es **nicht**, dass sich irgendwo eine `ModulName.psm1` oder `ModulName.psd1` befindet. Ein Modul muss folgende Struktur entsprechen:

```
ModulName\  
├─ ModulName.psm1  
└─ ModulName.psd1 (optional)
```

Vorraussetzungen:

- In einem der Verzeichnisse aus `$env:PSModulePath` muss ein **Ordner** mit dem Modulnamen existieren.
- In diesem Ordner muss sich eine gleichnamige Moduldatei (`.psm1`) oder ein gleichnamiges Manifest (`.psd1`) befinden.
- Dateien, die direkt im Modulpfad liegen (ohne entsprechenden Unterordner), werden bei `Import-Module ModulName` **nicht** berücksichtigt.

Manifest hat Vorrang

Existieren sowohl `ModulName.psd1` als auch `ModulName.psm1`, wird das **Manifest** importiert.

Dadurch können zusätzliche Informationen berücksichtigt werden, beispielsweise:

- weitere Module über `NestedModules`
- Abhängigkeiten über `RequiredModules`
- Exporte (`FunctionsToExport`, `AliasesToExport` usw.)
- Versions- und Kompatibilitätsinformationen

Deshalb ist ein Manifest der empfohlene Einstiegspunkt für größere Module.

☐ Import über Pfad

Wenn dein Modul irgendwo rumliegt, wo PowerShell es nicht findet:

```
Import-Module "C:\MeinProjekt\MeinModul.psm1"
```

Oder direkt das Verzeichnis:

```
Import-Module "C:\MeinProjekt\MeinModul"
```

PowerShell nimmt dann automatisch das [Manifest](#) (`.psd1`) oder die Moduldatei (`.psm1`).

☐ Erneutes Laden (Reload)

Du änderst Code, aber PowerShell hängt noch am alten Stand fest? Überraschung.

```
Import-Module MeinModul -Force
```

`-Force` lädt das Modul neu, auch wenn es schon geladen ist.

Pflicht beim Entwickeln, außer du stehst auf Selbstverwirrung.

☐ Selektiver Import

Du musst nicht alles laden. Du kannst gezielt nur bestimmte Dinge importieren:

```
Import-Module MeinModul -Function Get-Thing
```

Oder mehrere:

```
Import-Module MeinModul -Function Get-Thing, Set-Thing
```

Funktioniert natürlich nur, wenn dein Manifest das auch erlaubt (`FunctionsToExport`).

☐☐ Automatischer Import

Seit neueren PowerShell-Versionen gilt:

Wenn du eine Funktion aus einem Modul aufrufst, wird es automatisch importiert.

```
Get-Thing
```

→ PowerShell denkt sich: „Kenn ich nicht... ach warte, da gibt's ein Modul“

Klingt smart. Ist es auch.

Bis es nicht mehr funktioniert und du nicht weißt warum.

☐☐ Geladene Module anzeigen

```
Get-Module
```

Alle verfügbaren Module anzeigen:

```
Get-Module -ListAvailable
```

Wenn dein Modul hier nicht auftaucht, kannst du lange importieren. Wird nix.

☐☐ Modul entfernen

Falls du es wieder loswerden willst:

```
Remove-Module MeinModul
```

Hilfreich beim Entwickeln, wenn du nicht mit `-Force` arbeiten willst.

⚠ Bereits geladenes Modul

Wenn ein Modul bereits geladen ist und du erneut `Import-Module` aufrufst, passiert... nichts.

```
Import-Module MeinModul
```

PowerShell sagt sich: „Kenn ich schon“ und ignoriert dich stillschweigend.

Das bedeutet:

- Änderungen im Code werden **nicht** übernommen
- Du arbeitest weiter mit der alten Version

Wenn du sicherstellen willst, dass dein Modul wirklich neu geladen wird:

```
Import-Module MeinModul -Force
```

Alternativ kannst du es vorher entfernen:

```
Remove-Module MeinModul Import-Module MeinModul
```

Wichtig beim Entwickeln:

Wenn sich „irgendwas komisch anfühlt“, ist es erstaunlich oft genau das hier.

Du debugst dann nicht deinen Code.

Du debugst deine Vergangenheit.

Der eigentliche Punkt (den wieder alle ignorieren)

Import ist kein einmaliger Akt, sondern Teil deines Workflows.

- Entwicklung → ständig `-Force`
- Debugging → bewusst laden/entladen
- Produktion → einmal sauber laden und fertig

Wenn dein Modul sich beim Import komisch verhält, liegt es fast nie am Import.

Es liegt daran, was dein Modul beim Laden tut.

Und genau da trennt sich „läuft irgendwie“ von „ich hab verstanden, was ich da gebaut habe“.

Manifest (.psd1)

Was ist ein Manifest?

Ein Manifest (`.psd1`) beschreibt ein PowerShell-Modul.

Während die eigentliche Programmlogik in einer `.psm1`-Datei liegt, enthält das Manifest Informationen über das Modul selbst.

Dazu gehören beispielsweise

- Version
- Autor
- Abhängigkeiten
- exportierte Befehle
- unterstützte PowerShell-Versionen
- Lizenzinformationen

Das Manifest enthält **keinen ausführbaren Programmcode**. Es dient ausschließlich als Beschreibung und Konfiguration eines Moduls.

Allgemein

Das ist die Identität deines Moduls. Ohne das bist du einfach nur irgendein Skript mit Existenzkrise.

```
@{  
    RootModule      = "TestModul.psm1"  
    ModuleVersion    = "1.7.0"  
    GUID             = "e9e0799e-734e-451d-9bb1-34983dd82f05"  
    Author           = "ProgramMaster"  
    CompanyName      = "Microsoft"  
    Copyright        = "Microsoft i guess"  
    Description      = "Ich beschreibe wundervoll, was das für ein Modul es ist"  
}
```

RootModule

Hauptdatei (`.psm1` oder `.dll`)

ModuleVersion

Version (z. B. `1.0.0`)

GUID

Eindeutige ID

Author

Ersteller/Entwickler des Moduls

CompanyName

Unternehmen

Copyright

Urheberrechte

Description

Beschreibung des Moduls

Abhängigkeiten

Hier sagst du: „Ich funktioniere nicht alleine, ich brauche andere.“

- `RequiredModules`
 - Externe Abhängigkeiten
 - Angabe über **Modulnamen** (PowerShell sucht es selbst)
 - Müssen im System vorhanden sein (installiert / auffindbar, z.B. über `$PSModulePath`)
 - Gehören **nicht** zu deinem Modul
 - `RequiredAssemblies` – DLLs
 - `ScriptsToProcess` – Skripte, die beim Laden ausgeführt werden
 - `ModuleList` – Nur die Meta-Information welche andere Module zugehörig sind
 - `NestedModules`
 - Angabe über **Dateien/Pfade** (du sagst konkret, was geladen wird)
 - Interne Bausteine deines Moduls
 - Werden zusammen mit deinem Modul geladen
 - Gehören **zu deinem Modul dazu**
-

☐☐ Export (was nach außen sichtbar ist)

Das ist der Teil, wo du entscheidest, was du der Welt zeigst. Oder versteckst.

- `FunctionsToExport`
- `CmdletsToExport`
- `VariablesToExport`
- `AliasesToExport`

☐☐ Klassiker:

`'*'` = alles exportieren

oder explizit = bessere Kontrolle

☐☐ Kompatibilität & Anforderungen

Hier wird's picky.

- `PowershellVersion`
 - `PowershellHostName`
 - `PowershellHostVersion`
 - `DotNetFrameworkVersion`
 - `CLRVersion`
 - `ProcessorArchitecture`
-

☐☐ Private Daten (der geheime Keller)

Alles, was du selbst definierst.

- `PrivateData` – Hashtable für eigene Infos

Beispiel:

```
PrivateData = @{
    PSData = @{
        Tags = @('UI', 'Forms')
        ProjectUri = 'https://...'
    }
}
```

☐☐ PSData (wichtig für Gallery etc.)

Teil von `PrivateData`, aber so wichtig, dass es eigentlich sein eigenes Ding ist.

- `Tags`
- `LicenseUri`
- `ProjectUri`
- `IconUri`
- `ReleaseNotes`

☐☐ Sonstiges (der „Warum gibt’s das?“ Bereich)

Selten gebraucht, aber existiert halt:

- `FileList`
- `TypesToProcess`
- `FormatsToProcess`
- `CompatiblePSEditions` (`Desktop`, `Core`)
- `HelpInfoURI`

☐☐ Mini-Beispiel

Damit du nicht nur trockene Theorie hast:

```
@{
    RootModule      = 'MeinModul.psm1'
    ModuleVersion   = '1.0.0'
    GUID            = '12345678-abcd-1234-abcd-1234567890ab'
    Author          = 'Jonny'
    Description      = 'Mein erstes Modul'

    FunctionsToExport = @('Get-Thing', 'Set-Thing')

    PowerShellVersion = '5.1'

    PrivateData = @{
        PSData = @{
            Tags = @('Example', 'Test')
        }
    }
}
```

Der eigentliche Punkt (den viele übersehen)

Du brauchst vielleicht **30% davon wirklich**.

Der Rest ist:

- entweder für Publishing
- oder für Leute, die Kontrolle lieben (oder Angst haben)

Wenn du lokal entwickelst:

→ `RootModule`, `ModuleVersion`, `FunctionsToExport`

→ fertig, läuft.