

PowerShell for Admins

An introduction to PowerShell from a sysadmin's perspective

Agenda

- What's PowerShell & some history
- Basics
- Getting Help
- Providers
- Flow
- Remoting
- Building Tools

What have you done with PS?
What would you like to learn?

Thomas Torggler

- 10 years in IT
 - 2007-2009: Support, Italy
 - 2009-2013: Systems Administrator, Italy
 - 2013-2015: Senior Consultant (Unified Communications), Germany
 - 2015-current: Systems Engineer, Italy/Austria
- Cycling, Travelling, Languages
- Get in touch
 - <https://ntsystems.it/tom>

Housekeeping

- Feb. 03 – a 0/05
- Feb. 04 – a +1/03 (upstairs)
- Break: Whenever you need one ;-)
- Lunch: 12:30

So what is PowerShell?

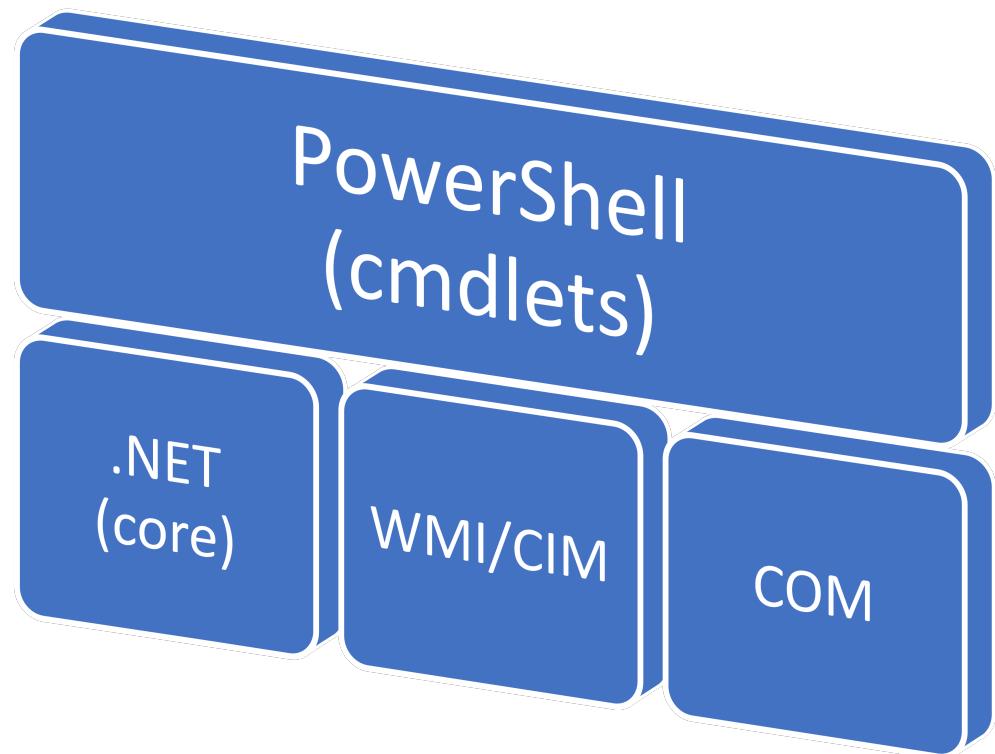
Definition

- Windows PowerShell is an interactive object-oriented command environment with scripting language features that utilizes small programs called **cmdlets** to simplify *configuration, administration, and management* of heterogeneous environments in both standalone and networked typologies by utilizing standards-based remoting protocols.

Source: <https://blogs.technet.microsoft.com/heyscriptingguy/2015/01/02/what-is-powershell/>

Definition (cont'd)

- .NET Framework Application
 - Security Model
- WMI/CIM Classes
- COM-Objects
- Interactive
- Scripting
- ...



The Monad Manifesto

- The original whitepaper by Jeffrey Snover (2002)
 - Problem: GUIs are not really suitable for automation; APIs are not easily consumable by Admins
 - Proposed Solution: A vital middle way, suitable for Admins (and Developers)
- Annotated version of the whitepaper:
<https://devopscollective.gitbooks.io/the-monad-manifesto-annotated/content/>
- V1 shipped with Windows Vista/Server 2008

Versions

- 2.0 Windows 7 / Server 2008R2
 - Remoting
 - Modules
- 3.0 Windows 8 / Server 2012
 - Jobs
 - Module Auto Loading
- 4.0 Windows 8.1 / Server 2012R2
 - DSC
- 5.1 Windows 10 Anniversary Update
 - OneGet (PackageManagement)
- 6.x.x Open Sourced; based on .NET Core
 - Nano Server
 - Linux, macOS

\$PSVersionTable

Basics

Interactive vs. Script

- The easiest way to get started
- Suitable for one-off tasks
- Few commands chained together
- Can easily be turned into a tool and re-used or shared!
 - Just save as *.ps1
 - Turn it into a function and create a module
 - Execution Policy (Get-, Set-)

Basic features

- 1 + 2
- (3GB + 4MB) / 1MB
- 1..9 ; 9..1
- () are evaluated first
- Escape Character: ` (backtick)
- Single vs. Double Quotes
 - 'a string' is "a string" but...
 - „It's \$((Get-Date).TimeOfDay)“
- Aliases (to avoid): %, ?, dir, cat...

History

- Command History is stored and can be reused
- **\$MaximumHistoryCount**
 - Controls how many entries are stored

Get-History

Invoke-History –Id xx

Transcript

- Easy documentation ;-)
- Start-Transcript
 - Looks for \$Global:Transcript
 - Path parameter
 - Documents folder
- Stop-Transcript

```
New-Variable -Scope Global -Name Transcript -  
Value documentation.txt
```

Cmdlets & Parameters

- Verbs
 - Well-known, approved
 - Get-Verb
- Nouns
 - Singular, specific, descriptive
 - Pascal Case (Get-ChildItem)
- Use **Get-Command** to discover cmdlets by verb, noun, module...
- Parameters
 - Input
 - Singular, standard (Name, not ItemName)
 - Validation

Autocomplete with “tab”

Common Parameters

- Built-in parameters for all cmdlets
- Cannot be used in custom functions
- General
 - `Debug`, `ErrorAction`, `ErrorVariable`, `OutBuffer`,
`OutVariable`, `WarningAction`, `WarningVariable`, `Verbose`
- Risk Mitigation
 - `WhatIf`, `Confirm`
- Transaction
 - `UseTransaction`

Help & Discovery

Updateable Help?

- PowerShell is a part of Windows and can only be updated by “Service Packs” (pre Win10)...
- The PowerShell Team required a faster method to ship help files
- Run: `Update-Help -UICulture en-us`
- Put it into your `$Profile`
`Start-Job -Name UpdateHelp ``
`-Command {Update-Help -UICulture en-us}`

Asking for Help

- `help Get-Process`
 - One page at a time
- `Get-Help Get-Process -ShowWindow`
 - Opens a window with searchbox
- `Get-Help Get-Process -Parameter Name`
 - Show help for a specific parameter
- `Get-Help Get-Process -Online`
 - Show online version of the help content (if available)

Help Files

- `Get-Help about_Arrays`
 - Shows help file for Arrays
- `Get-Help -Category HelpFile`
 - Shows all Help Files (`about_*`)

Discovering commands

- Google? Stack Overflow? GitHub? ;-)
 - Do “x” with PowerShell
- `Get-Module -ListAvailable`
- `Get-Command -Module PackageManagement`
- `Get-Command -Parameter CimSession`
- ISE
- `Show-Command -Name New-Item`

Variables

Variables

- Stored in the **variable**: drive
- Store values that can be used, typically command outputs
- Assigned with “=” character

```
$a = 1
```

```
$b = "a","b","c"
```

```
$procs = Get-Process
```

- Other assignment operators

```
+ =, --, *=, /=, %=, ++, --
```

Get-Help about_Variables

Variables (cont'd)

- Arrays
 - Collections
- \$array = @(1,2,3)
- Index using number: \$array[1]
- Add items using += which creates a new array

[Get-Help about_Arrays](#)

Variables (cont'd)

- Hash Tables (dictionary)

- Key=Value pairs

```
$ht = @{
    Name = "Thomas";
    Language = "PowerShell";
    Age = 28;
}
```

- Index using the key: `$ht.Name`
- Add/Remove using the respective Method: `$ht.Remove("Age")`

[Get-Help about_Hash_Tables](#)

Automatic Variables

- **\$?**
 - Execution status of last operation
 - True if succeeded; False if failed
- **\$True and \$False**
 - True and False ;-)
- **\$_ or \$PSItem**
 - Contains the current object in the pipeline
- **\$PSVersionTable, \$PSEdition**
- **\$PSCmdlet, \$PSBoundParameters...**

[Get-Help about_Automatic_Variables](#)

Preference Variables

- Can be used to customize the behavior of PowerShell
- **\$ConfirmPreference**
 - High impact cmdlets (Remove*) should have a Confirm parameter
- **\$WarningPreference**
 - Default is to display warning and continue execution
- **\$WhatIfPreference**
 - Default is \$false; set to \$true to use WhatIf by default
- **\$ErrorActionPreference**
 - Default is to display error and continue execution for non terminating errors

[Get-Help about_Preference_Variables](#)

Objects & Data Types

Objects and the Pipeline

- PowerShell cmdlets return *Objects*
- Objects have *Members*
- Members include *Properties* and *Methods*
- Objects returned by a cmdlet can be used as input for another cmdlet
 - `Get-Process | Where-Object Name -like "WiFi*`
 - In this case, Get-Process returns all processes and Where-Object is used to filter for a specific Name.
 - Remember: Name is a *Property*, use Get-Member to list available members

`Get-Help about_Objects`

Objects (cont'd)

- `Get-Command -Noun Object`
 - `Select-;` `Where-;` `Measure-;` `Sort-;` `Group-...`
- Can be created easily from a hash table

```
$ht = @{
    Name = "Thomas";
    Language = "PowerShell";
    Age = 28;
}
$obj = New-Object -Property $ht -TypeName
pscustomobject
```

Objects (cont'd)

- Import-* / Export-*
- Out-*
- ConvertTo-* / ConvertFrom-*
- Format-*
- Extract nested properties using
| Select-Object Name,@{Name="MyName";
Expression={\$_.Nested.Property}}}

Data Types

- Data type is determined by .NET
- Generally loosely typed

\$a = 1

\$b = "2"

\$a + \$b = ?

\$b + \$a = ?

[int]\$b + \$a = ?

\$a.GetType()
"2" | Get-Member

- Operators
 - is, -isNot, -as

Compare

```
"1" -ne "2"  
1 -is [int]  
(1..2) -is [Array]  
"PowerShell" -match „P.*“; $Matches
```

- Note: „=“ is an assignment operator and is not used for comparison

```
$a = 1 # Set the content of $a to 1  
$a -eq 1 # Compare the content of $a with 1
```

[Get-Help about_Comparison_Operators](#)

\$Profile

Profile

- A script that gets loaded every time PowerShell starts
- **\$PSHOME\profile.ps1**
This profile applies to all users and all shells.
- **\$PSHOME\Microsoft.PowerShell_profile.ps1**
This profile applies to all users, but only to the Microsoft.PowerShell shell.
- **\$HOME\Documents\WindowsPowerShell\profile.ps1**
This profile applies only to the current user, but affects all shells.
- **\$HOME\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1**
This profile applies only to the current user and the Microsoft.PowerShell shell.

Profiles Linux/macOS

- **\$PSHOME/profile.ps1**

This profile applies to all users and all shells.

- **\$PSHOME/Microsoft.PowerShell_profile.ps1**

This profile applies to all users, but only to the Microsoft.PowerShell shell.

- **\$HOME/.config/powershell/profile.ps1**

This profile applies only to the current user, but affects all shells.

- **\$HOME/.config/powershell/Microsoft.PowerShell_profile.ps1**

This profile applies only to the current user and the Microsoft.PowerShell shell.

Profile example

- Default values for parameters can be set using a preference variable

```
$PSDefaultParameterValues=@{"<CmdletName>:<Parameter  
Name>"=<DefaultValue>"}
```

```
$PSDefaultParameterValues = @{"Send-  
MailMessage:SmtpServer"="mail01"}  
$PSDefaultParameterValues = @{"*:Verbose"=$true}
```

```
$PSDefaultParameterValues.Add("*:Verbose", $true)
```

Try it yourself

- Create a profile and customize your session
- Check out `$Profile` and try `Test-Path` and `New-Item`

Modules

Modules

- A set of PowerShell functionalities grouped together as a unit (folder)
- Enables modularization ;)
 - Reuse and abstraction of code
 - Publish functions
 - Define scope for functions/variables
- The Module Manifest (*.psd1) contains a hash table and defines
 - The contents and attributes of the module
 - The prerequisites
 - How the components are processed

Modules (cont'd)

- Script Module
 - Consists of one or more plaintext (*.psm1) files
 - Written in PowerShell
- Binary Module
 - .NET framework assembly (*.dll)
 - Written e.g. in C#
- Dynamic Module
 - Not saved to a file
 - Created by script or remoting
 - Intended to be short-lived/non-persistent

Modules (cont'd)

- **Get-Module**
 - List imported modules
 - -ListAvailable lists available modules
- **Import-Module**
 - Import a module, i.e. make it available in the current session
 - Remember auto-loading (>3.0)
- **\$PSModuleAutoLoadingPreference**
 - Control module auto-loading behaviour
- **\$env:PSModulePath**
 - A list of directories containing PowerShell Modules

Modules (cont'd)

- User modules

\$HOME\Documents\WindowsPowerShell\Modules

- Shared modules will be read from

\$env:ProgramFiles\WindowsPowerShell\Modules

- Default modules will be read from

\$PSHOME\Modules

```
$env:PSModulePath -split ";"
```

Modules Linux/macOS

- User modules

\$HOME/.local/share/powershell/Modules

- Shared modules will be read from

/usr/local/share/powershell/Modules

- Default modules will be read from

\$PSHOME/Modules

```
$env:PSModulePath -split ":"
```

Providers

Providers and Drives

- Provide access to data/components that would not otherwise be easily accessible
- Data is shown as a file system drive
- Get-PSProvider lists available providers
- Try:

```
Get-ChildItem Cert:  
cd hklm: ; dir  
cd hkcu: ; dir
```

Providers (cont'd)

- Default, built-in providers
- Can be expanded by Modules or 3rd parties
 - Active Directory
 - VMware

```
Get-ChildItem Cert:\LocalMachine\My
```

```
Get-ItemProperty  
'HKLM:\SOFTWARE\Microsoft\Windows  
NT\CurrentVersion' -Name CurrentVersion
```

Controlling Flow

If, Elseif, Else

- Evaluates a condition and executes code if condition is met

```
If ($x -eq 1) {  
    # executes if $x is 1  
} elseif ($x -eq 2) {  
    # executes if $x is 2 (but not 1)  
} else {  
    # executes if $x is any other value  
}
```

[Get-Help about_If](#)

Switch

- Handle multiple if statements

```
switch (4, 2) {  
 1 {"One." }  
 2 {"Two." }  
 3 {"Three." }  
 4 {"Four."; Break }  
 2 {"Two again."}  
}
```

- All statements are evaluated unless there is a ; Break
- Default triggers when no other condition matches

[Get-Help about_Switch](#)

ForEach

- Loops through a collection of objects

```
ForEach ($i in 1..9) {  
    $i  
}
```

```
1..9 | ForEach-Object {  
    $PSItem  
}
```

[Get-Help about_Foreach](#)

While (<condition>) {<command>}

- Execute statement while condition is \$true

```
while($true) {  
    "this will loop until stopped with ctrl-c"  
}
```

```
$i = 0  
while($i -lt 9) {  
    $i++  
    "`$i is $i"  
}
```

[Get-Help about_While](#)

For (<init>;<cond>;<rep>) {<command>}

- Execute statement while condition is \$true
- Typically used to iterate through arrays; prefer ForEach

```
$var = 1..9
for ($i = 0; $i -lt $var.count; $i++) {
    "`$i is $($var[$i])"
}
```

```
$var.ForEach{"`$i is $PSItem"}
```

Get-Help about_For

Remoting

Remoting

- Run commands on one or more remote computers and get output in local session

- ComputerName Parameter

```
Get-EventLog –ComputerName s01,s02
```

- Interactive with Enter-PSSession

```
Enter-PSSession s01
```

```
S01\PS>...
```

Remoting (cont'd)

- One-to-many

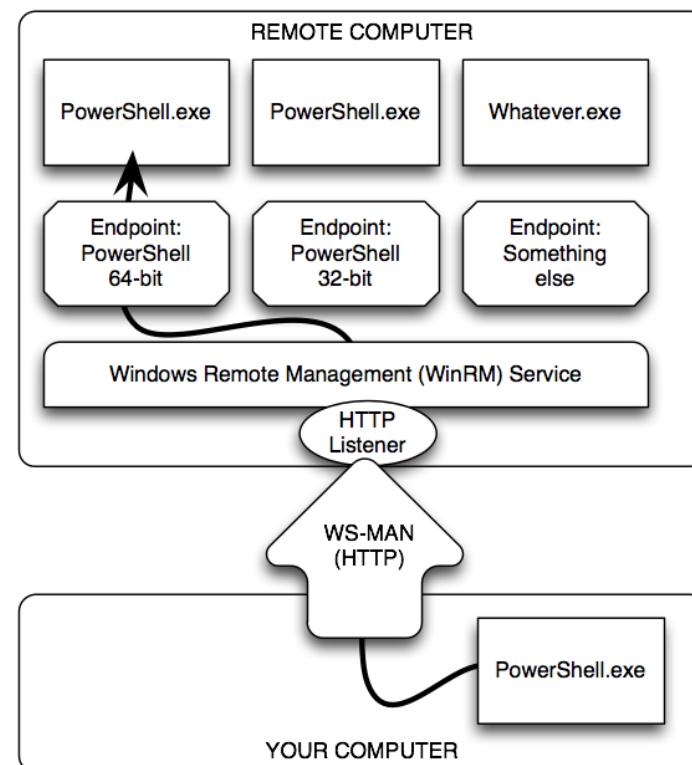
```
Invoke-Command -ComputerName s01,s02 -ScriptBlock { #  
do something }
```

- Sessions for persistence

```
$s = New-PSSession -ComputerName s01  
Invoke-Command -Session $s -ScriptBlock { # do  
something}
```

Remoting (cont'd)

- Commands are sent to remote machine
- Remote machine executes command
- Output is serialized to XML and sent back
- Local session deserializes XML and displays information
 - No methods are available locally



Remoting security

- Requires membership of “Administrators” group on the target machine
- Session configuration (endpoint) can be used to configure the session parameters
- Uses encryption and delegation (credentials are never sent to remote machine)
- Mutual authentication is required by default (domain)
- https (certificate based authentication) can be used for non-domain or cross-domain scenarios

PSSession vs. CimSession

- Both use encryption over tcp/5985

```
$pss = New-PSSession -ComputerName server101  
$cim = New-CimSession -ComputerName server 101
```

```
Invoke-Command -Session $pss -ScriptBlock {Get-Disk}
```

```
Get-Disk -CimSession $cim | Set-Disk...
```

Import Session

- A dynamic module is created
- Remote commands are made available in local session

```
$s = New-PSSession -ComputerName server101  
Import-PSSession $s -CommandName Get-Process
```

Error Handling

Try and Catch

- A terminating error inside a `try {}` block can be handled gracefully by whatever is defined in the `catch {}` block

```
Try {  
    SomethingThatCouldBreak  
}  
  
Catch {  
    "handle gracefully and continue execution"  
}
```

[Get-Help about_Try_Catch_Finally](#)

Finally

- Always executes (even when exit, ctrl-c)
- Can be used to cleanup resources
- `$Error` contains error messages (latest is [0])
- Don't override `$ErrorActionPreference` globally
 - If really necessary it can be done using the `-ErrorAction` parameter

Throw

- Generates a terminating error
- If used in catch block (without further parameters) echoes the exception

```
throw "That did not work!"
```

```
throw $PSItem
```

Background Jobs

Background Jobs

- Long-running tasks can be sent to the background to make the command prompt available for use
- **Start-Job -ScriptBlock { Get-Process }**
 - Starts a background job in current session
- **Get-Job**
 - Gets background jobs that were started in current session
- **Receive-Job**
 - Returns data only once, make sure to store in variable or export

Building Tools

```
function Verb-Noun {...
```

- Specific, do one thing and do it right
- Validate Inputs
- Do something
- Output Objects to the pipeline

```
...}
```

[CmdletBinding()]

- Enables Common Parameters
- Confirm Preference
- Should Process
- #Requires statements

[Get-Help about_Requires](#)

param()

- Input validation
 - [ValidateNotNull()]
 - [ValidateSet(1,2,3)]
- Accept pipeline input
 - [Parameter(Mandatory=\$true,Position=1)]
 - [Parameter(ValueFromPipeline=\$true)]

Write-Verbose

- Write information to verbose stream if Verbose parameter is used
- Useful for power users or troubleshooting
- `Write-Debug -Message`
- Redirect with 4>
 - `New-Item -Name test1 -Verbose 4> Verbose.log`

Write-Debug

- Pauses execution and writes debug output to the console if Debug parameter is used
- Useful for troubleshooting, step-by-step execution
- Write-Debug -Message
- Redirect with 5>
 - New-Item -Name test1 -Debug 5> Debug.log

Misc

Scope

- Items are created in a scope to limit where they can be accessed and changed
- Child scope does not inherit but can access items

```
function Test-Scope {  
    $test = 1 ; $test  
    function Test-ChildScope {  
        $test2 = 2; $test; $test2  
    }  
    Test-ChildScope  
}
```

Resources

- PowerShell Home: <https://msdn.microsoft.com/en-us/powershell/>
- Free eBooks: <https://www.gitbook.com/@devopscollective>
- <https://www.powershellgallery.com>
- <https://powershell.org>
- Documents and snippets:
<https://github.com/tomtorggler/PoSh4Admins>