PowerShell

… EN DESARROLLO …

Introducción

PowerShell es una solución de automatización de tareas multiplataforma formada por un **shell de línea de comandos**, un **lenguaje de scripting** y un **marco de administración de configuración**. PowerShell está diseñado por Microsoft y funciona en Windows 10, Linux y macOS.

Introducido por primera vez en 2006, PowerShell ha evolucionado hasta convertirse en una herramienta esencial para administradores de sistemas, desarrolladores y profesionales de IT.

A diferencia de las interfaces tradicionales de línea de comandos, PowerShell utiliza un enfoque basado en objetos, lo que significa que las salidas de los comandos son representadas como objetos estructurados que pueden ser manipulados y filtrados de manera intuitiva. Esto proporciona una gran flexibilidad y eficiencia al realizar tareas administrativas y permite una mayor automatización de procesos.

Algunas de las características de PowerShell son:

- **Cmdlets**: PowerShell utiliza "*cmdlets*" (comandos pequeños) como unidades básicas de trabajo. Estos *cmdlets* realizan funciones específicas y se pueden encadenar para crear secuencias más complejas de comandos.
- **Interactividad y scripts**: Se puede utilizar PowerShell de manera interactiva en la línea de comandos o escribir scripts completos para automatizar tareas repetitivas. Los scripts permiten realizar acciones complejas con un solo comando.
- **Acceso a tecnologías Microsoft**: PowerShell ofrece acceso directo a las tecnologías de Microsoft, como el Active Directory, Exchange, SharePoint y otras, facilitando la administración de entornos empresariales.
- **Consola gráfica**: además de la interfaz de línea de comandos, PowerShell cuenta con una consola gráfica llamada PowerShell ISE (Integrated Scripting Environment) que facilita la creación, depuración y ejecución de scripts.
- **Soporte multiplataforma**: a partir de la versión PowerShell Core, la herramienta se ha vuelto multiplataforma, lo que significa que ahora está disponible para sistemas operativos como Linux y macOS, ampliando su utilidad más allá del ecosistema Windows.
- **Automatización más eficiente**: PowerShell simplifica tareas administrativas mediante la automatización de procesos, ahorrando tiempo y reduciendo posibles errores humanos.

- **Consistencia**: la estructura basada en objetos y la sintaxis coherente de PowerShell contribuyen a una administración consistente y predecible de sistemas.

- **Integración con .NET**: al estar construido sobre el marco .NET, PowerShell permite la integración con bibliotecas y servicios .NET, ampliando aún más sus capacidades.

En resumen, PowerShell es una herramienta poderosa que ofrece una forma eficaz de administrar y automatizar tareas en entornos Windows y más allá, convirtiéndose en una herramienta esencial para profesionales de IT y administradores de sistemas.

> Sección generada por ChatGPT

Windows PowerShell vs PowerShell Core

Windows PowerShell es la implementación original de PowerShell lanzada por Microsoft junto con Windows 7 en 2009. Está diseñada específicamente para entornos Windows y utiliza el marco .NET Framework. A lo largo de los años, Windows PowerShell se convirtió en una herramienta esencial para la administración de sistemas Windows, proporcionando una interfaz de línea de comandos y un entorno de scripting robusto para la automatización de tareas.

Por otro lado, **PowerShell Core** es una versión posterior y más avanzada de PowerShell que fue lanzada en 2016. La principal diferencia es que PowerShell Core es **multiplataforma**, lo que significa que puede ejecutarse no solo en sistemas operativos Windows, sino también en Linux y macOS. Además, PowerShell Core es de código abierto y se basa en el marco .NET Core, permitiendo una mayor flexibilidad y portabilidad.

- **Ecosistema común**: ambas versiones comparten un conjunto básico de conceptos y cmdlets, lo que facilita la transición entre ellas. Sin embargo, hay algunas diferencias en los cmdlets y características específicas de cada versión.

- **Compatibilidad**: aunque PowerShell Core es más versátil al ser multiplataforma, Windows PowerShell sigue siendo la opción predeterminada en entornos Windows tradicionales. Muchos scripts y comandos desarrollados en Windows PowerShell pueden ejecutarse en PowerShell Core, pero puede haber casos en los que se requieran ajustes debido a las diferencias de plataforma.

- **Desarrollo continuo**: Microsoft ha indicado que PowerShell Core es el camino a seguir y recibirá actualizaciones continuas, mientras que Windows PowerShell está en modo de mantenimiento. A medida que evoluciona el ecosistema, se alienta a los usuarios a adoptar PowerShell Core para aprovechar las nuevas características y mejoras.

En resumen, PowerShell Core representa la evolución de Windows PowerShell, siendo más versátil y adaptable a diferentes plataformas, mientras que Windows PowerShell sigue siendo relevante en entornos Windows más tradicionales. La elección entre ambas versiones dependerá de los requisitos específicos del entorno y la plataforma.

> Sección generada por ChatGPT

Resumen

```
Get-ExecutionPolicy -List
Set-ExecutionPolicy AllSigned
# Otras opciones de políticas de ejecución son:
# - Restricted: Los scripts no correrán.
# - RemoteSigned: Los scripts que se hayan descargado sólo correrán si han sido firmados por un editor de confianza.
# - AllSigned: Los scripts requieren ser firmados por un editor de confianza.
# - Unrestricted: Ejecuta cualquier script.
help about_Execution_Policies # para obtener más ayuda sobre políticas de ejecución.

# Versión instalada de PowerShell:
$PSVersionTable

# Si necesita encontrar algún comando
```

```powershell
Get-Command about_* # tiene por abreviación (o alias): gcm
Get-Command -Verb Add # lista todos los comandos que tienen por verbo 'Add'
Get-Alias ps # Lista el alias de ps
Get-Alias -Definition Get-Process

Get-Help ps | less # alias: help
ps | Get-Member # alias: gm

Show-Command Get-EventLog # Muestra un formulario para llenar los parámetros del comando Get-EventLog

Update-Help # Actualiza la ayuda (debe ser ejecutado en una consola elevada como admin)

# Como ya lo notó, los comentarios empiezan con #

# Ejemplo de un simple hola mundo:
Write-Output Hola mundo!
# echo es el alias del comando Write-Output (a los comandos también se les dice cmdlets)
# La mayoría de los cmdlets y funciones siguen la convención de llamarse de la forma: Verbo-Sustantivo

# Cada comando inicia en una nueva línea, o después de un punto y coma:
Write-Output 'Esta es la primer línea'; echo 'Esta es la segunda'

# La declaración de una variable se ve así:
$unaCadena ="Algún texto"
# O así:
$unNumero = 5 -as [double]
$unaLista = 1,2,3,4,5
$unaCadena = $unaLista -join '--' # también existe el parámetro -split
$unaTablaHash = @{nom1='val1'; nom2='val2'}

# Uso de variables:
echo $unaCadena
echo "Interpolación: $unaCadena"
echo "`$unaCadena tiene longitud de $($unaCadena.Length)"
echo '$unaCadena'
echo @"
Esta es una Here-String
$otraVariable
"@
# Note que una ' (comilla simple) no expande las variables!
# Las Here-Strings también funcionan con comilla simple

# Variables Automáticas:
# Hay algunas variables previamente definidas en el ambiente que le pueden servir, tales como
echo "Booleanos: $TRUE y $FALSE"
echo "Valor vacío: $NULL"
echo "Valor de retorno del último programa: $?"
echo "Código de salida del último programa en Windows: $LastExitCode"
echo "El último token en la última línea de la sesión activa: $$"
echo "El primer token: $^"
echo "PID del script: $PID"
echo "Ruta completa del directorio dónde está el script actual: $PSScriptRoot"
echo 'Ruta completa de script actual: ' + $MyInvocation.MyCommand.Path
echo "Ruta completa de directorio actual: $Pwd"
echo "Argumentos pasados a la invocación de una función, script o bloque de código: $PSBoundParameters"
echo "Argumentos no predefinidos: $($Args -join ', ')."
# Para saber más sobre variables automáticas: `help about_Automatic_Variables`

# Para enlazar otro archivo (operador punto)
. .\otroNombreDeScript.ps1


### Control de Flujo
# Tenemos la estructura de if como es usual:
if ($Edad -is [string]) {
  echo 'Pero... si $Edad no puede ser una cadena de texto!'
} elseif ($Edad -lt 12 -and $Edad -gt 0) {
  echo 'Niño (Menor de 12. Mayor que 0)'
} else {
  echo 'Adulto'
}
```

```powershell
# Sentencias switch de PS son más poderosas comparadas con otros lenguajes
$val = "20"
switch($val) {
  { $_ -eq 42 }           { "La respuesta es 42"; break }
  '20'                    { "Exactamente 20"; break }
  { $_ -like 's*' }       { "No distingue entre mayúsculas/minúsculas"; break }
  { $_ -clike 's*'}       { "clike, ceq, cne para ser diferenciar el caso entre mayúsculas/minúsculas"; break }
  { $_ -notmatch '^.*$'}  { "Emparejamiento de expresiones regulares. cnotmatch, cnotlike, ..."; break }
  { 'x' -contains 'x'}    { "FALSO! -contains es para listas!"; break }
  default                 { "Otros" }
}

# El for clásico
for($i = 1; $i -le 10; $i++) {
  "Número de ciclo $i"
}
# O más corto
1..10 | % { "Número de ciclo $_" }

# PowerShell también incluye
foreach ($var in 'valor1','valor2','valor3') { echo $var }
# while () {}
# do {} while ()
# do {} until ()

# Manejo de excepciones
try {} catch {} finally {}
try {} catch [System.NullReferenceException] {
 echo $_.Exception | Format-List -Force
}


### Proveedores
# Lista de archivos y directorios en la ubicación actual
Get-ChildItem # o el alias `dir`
Set-Location ~ # ir al directorio principal del usuario

Get-Alias ls # -> Get-ChildItem
# ¿¡Eh!? Estos cmdlets tienen nombres genéricos porque a diferencia de otros lenguajes de scripting,
# PowerShell no opera únicamente en el directorio actual.
Set-Location HKCU: # se dirige a la rama HKEY_CURRENT_USER del registro de Windows

# Para hacer un listado de todos los proveedores disponibles
Get-PSProvider


### Tuberías
# Los Cmdlets tienen parámetros que controlan su ejecución:
Get-ChildItem -Filter *.txt -Name # Se obtiene sólo el nombre de todos los archivos txt
# Sólo se necesita escribir caracteres de un parámetro hasta que deja de ser ambiguo
ls -fi *.txt -n # -f no se puede porque también existe -Force
# Use `Get-Help Get-ChildItem -Full` para un tratado más completo

# los results del cmdlet anterior se le pueden pasar como entrada al siguiente.
# `$_` representa el objeto actual en el objeto de tubería.
ls | Where-Object { $_.Name -match 'c' } | Export-CSV exportado.txt
ls | ? { $_.Name -match 'c' } | ConvertTo-HTML | Out-File exportado.html

# Si se confunde con la tubería use `Get-Member` para revisar
# los métodos y propiedades de los objetos de la tubería:
ls | Get-Member
Get-Date | gm

# ` es el caracter de continuación de línea. O termine la línea con un |
Get-Process | Sort-Object ID -Descending | Select-Object -First 10 Name,ID,VM `
 | Stop-Process -WhatIf

Get-EventLog Application -After (Get-Date).AddHours(-2) | Format-List

# Use % como una abreviación de ForEach-Object
```

```powershell
(a,b,c) | ForEach-Object `
 -Begin { "Iniciando"; $counter = 0 } `
 -Process { "Procesando $_"; $counter++ } `
 -End { "Terminando: $counter" }

# El siguiente comando ps (alias de Get-Process) devuelve una tabla con 3 columnas
# La tercera columan es el valor de memoria virtual en MB y usando 2 dígitos decimales
# Las columnas calculadas pueden escribirse más extensamente como:
# `@{name='lbl';expression={$_}`
ps | Format-Table ID,Name,@{n='VM(MB)';e={'{0:n2}' -f ($_.VM / 1MB)}} -autoSize


### Funciones
# El atributo [string] es opcional.
function foo([string]$nombre) {
 echo "Hey $nombre, aquí tiene una función"
}

# Llamando una función
foo "Diga mi nombre"

# Funciones con parámetros nombrados, atributos de parámetros y documentación analizable
<#
.SYNOPSIS
Establecer un nuevo sitio web
.DESCRIPTION
Crea todo lo que su sitio necesite
.PARAMETER siteName
El nombre para el nuevo sitio web
.EXAMPLE
Crear-SitioWeb -Nombre SitioBonito -Po 5000
Crear-SitioWeb SiteWithDefaultPort
Crear-SitioWeb nombreSitio 2000 # ERROR! No se pudo validar arguemento de puerto
('nombre1','nombre2') | Crear-SitioWeb -Verbose
#>
function Crear-SitioWeb() {
 [CmdletBinding()]
 param (
  [Parameter(ValueFromPipeline=$true, Mandatory=$true)]
  [Alias('nombre')]
  [string]$nombreSitio,
  [ValidateSet(3000,5000,8000)]
  [int]$puerto = 3000
 )
 BEGIN { Write-Verbose 'Creando nuevo(s) sitio(s) web' }
 PROCESS { echo "nombre: $nombreSitio, puerto: $puerto" }
 END { Write-Verbose 'Sitio(s) web creado(s)' }
}


### Todo es .NET
# Una cadena PS es, de hecho, una cadena tipo System.String de .NET
# Todos los métodos y propiedades de .NET están disponibles
'cadena'.ToUpper().Replace('E', 'eee')
# O más powershellezco
'cadena'.ToUpper() -replace 'E', 'eee'

# ¿No recuerda cómo es que se llama cierto método .NET?
'cadena' | gm

# Sintaxis para ejecutar métodos .NET estáticos
[System.Reflection.Assembly]::LoadWithPartialName('Microsoft.VisualBasic')

# Nótese que cualquier función que proviene de .NET Framework REQUIERE paréntesis para ser invocada
# al contrario de las funciones definidas desde PS, las cuales NO PUEDEN ser invocadas con paréntesis.
# Si se invoca una función/cmdlet de PS usando paréntesis,
# es equivalente a que le estuviera pasando un parámetro de tipo lista
$writer = New-Object System.IO.StreamWriter($ruta, $true)
$writer.Write([Environment]::NewLine)
$writer.Dispose()
```

```powershell
### Entrada/Salida
# Leyendo una variable
$Nombre = Read-Host "¿Cómo se llama?"
echo "¡Hola $Nombre!"
[int]$Edad = Read-Host "¿Cuál es su edad?"

# Test-Path, Split-Path, Join-Path, Resolve-Path
# Get-Content filename # devuelve un string[]
# Set-Content, Add-Content, Clear-Content
Get-Command ConvertTo-*,ConvertFrom-*


### Material útil
# Actualizar la ruta de ejecuciones (PATH)
$env:PATH = [System.Environment]::GetEnvironmentVariable("Path", "Machine") +
 ";" + [System.Environment]::GetEnvironmentVariable("Path", "User")

# Encontrar Python en el path
$env:PATH.Split(";") | Where-Object { $_ -like "*python*"}

# Cambiar el directorio de trabajo sin tener que memorizar la ruta anterior
Push-Location c:\temp # se cambia el directorio de trabajo a c:\temp
Pop-Location # revierte el cambio y se devuelve a donde estaba al principio
# Los aliases son : pushd y popd

# Desbloquear un archivo después de descargarlo de Internet
Get-ChildItem -Recurse | Unblock-File

# Abre Windows Explorer en la ruta actual (usando el alias ii de Invoke-Item)
ii .

# Pulse cualquier tecla para salir
$host.UI.RawUI.ReadKey()
return

# Para crear un acceso directo
$WshShell = New-Object -comObject WScript.Shell
$Shortcut = $WshShell.CreateShortcut($link)
$Shortcut.TargetPath = $file
$Shortcut.WorkingDirectory = Split-Path $file
$Shortcut.Save()

# $Profile es la ruta completa para su `Microsoft.PowerShell_profile.ps1`
# Todo el código alojado allí será ejecutado cuando se ejecuta una nueva sesión de PS
if (-not (Test-Path $Profile)) {
 New-Item -Type file -Path $Profile -Force
 notepad $Profile
}
# Más información en: `help about_profiles`
# Para un shell más productivo, asegúrese de verifivar el proyecto PSReadLine descrito abajo
```

Enlaces de interés

- https://learn.microsoft.com/es-es/powershell/
- https://www.powershellgallery.com/
- https://code.visualstudio.com/docs/languages/powershell
- https://github.com/janikvonrotz/awesome-powershell
- https://gist.github.com/pcgeek86/336e08d1a09e3dd1a8f0a30a9fe61c8a
- https://www.maquinasvirtuales.eu/curso-basico-de-powershell-introduccion/

Licencia