PowerShell Essencial para Profissionais de TI

Autor: Paulo César Marino

1. INTRODUÇÃO AO POWERSHELL

1.1 História, Evolução e Posicionamento no Ecossistema Microsoft

1.1.1 Contexto Histórico e Necessidade de Criação

O PowerShell nasceu de uma necessidade crítica da Microsoft em oferecer uma ferramenta robusta de automação e administração para seus sistemas operacionais. Antes de sua criação, os administradores de sistemas Windows dependiam principalmente de:

- **Prompt de Comando (CMD)**: Interface de linha de comando limitada, herdada do MS-DOS
- **Scripts VBScript e JScript**: Soluções fragmentadas e sem consistência
- **Ferramentas GUI**: Interfaces gráficas que não permitiam automação eficiente
- **Windows Management Instrumentation (WMI)**: Poderoso, mas complexo e difícil de usar

Enquanto sistemas Unix/Linux possuíam shells poderosos como Bash, Ksh e Zsh há décadas, o Windows carecia de uma solução unificada e moderna para administração via linha de comando.

1.1.2 Linha do Tempo do PowerShell

2002-2003: Projeto Monad

- Iniciativa liderada por Jeffrey Snover, arquiteto da Microsoft
- Objetivo: criar um shell orientado a objetos para Windows
- Nome de código: "Monad"
- Conceito revolucionário: trabalhar com objetos .NET ao invés de texto puro

Novembro de 2006: PowerShell 1.0

- Lançamento oficial como componente opcional do Windows
- Baseado no .NET Framework 2.0
- Aproximadamente 130 cmdlets nativos
- Limitações: sem suporte para remoting nativo, módulos limitados

Outubro de 2009: PowerShell 2.0

- Integrado ao Windows 7 e Windows Server 2008 R2
- Recursos revolucionários:
    - **PowerShell Remoting**: Administração remota via WS-Management
    - **Background Jobs**: Execução assíncrona de comandos
    - **Advanced Functions**: Funções com capacidades de cmdlets
    - **PowerShell ISE**: Ambiente de script integrado
    - **Módulos**: Sistema de empacotamento e distribuição de código

Agosto de 2012: PowerShell 3.0

- Lançado com Windows 8 e Windows Server 2012
- Melhorias significativas:
    - **Workflow**: Integração com Windows Workflow Foundation
    - **CIM Cmdlets**: Sucessores dos cmdlets WMI
    - **Sessões desconectadas**: Resiliência em conexões remotas
    - **Descoberta automática de módulos**
    - **Intellisense aprimorado**

Outubro de 2013: PowerShell 4.0

- Incluído no Windows 8.1 e Windows Server 2012 R2
- Recursos principais:
    - **Desired State Configuration (DSC)**: Gerenciamento declarativo de configurações
    - **Melhorias no debugging**: Depuração de scripts e workflows
    - **Cmdlets para gerenciamento de rede aprimorados**

Fevereiro de 2016: PowerShell 5.0

- Última versão do "Windows PowerShell"
- Recursos marcantes:
    - **PowerShellGet**: Gerenciador de pacotes para módulos
    - **Classes**: Suporte nativo para programação orientada a objetos

    - **Enumerações e validações avançadas**
    - **OneGet/PackageManagement**: Framework unificado de gerenciamento de pacotes
    - **Melhorias significativas no DSC**

Janeiro de 2018: PowerShell Core 6.0

- **Mudança de paradigma**: Migração para .NET Core
- **Multiplataforma**: Suporte oficial para Linux e macOS
- **Open Source**: Código aberto no GitHub
- **Renomeação**: "Windows PowerShell" (5.1) vs "PowerShell Core" (6.x+)
- Objetivo: unificar a experiência de automação em todos os sistemas operacionais

Setembro de 2018: PowerShell Core 6.1

- Melhorias de performance e compatibilidade
- Suporte aprimorado para módulos do Windows PowerShell

Janeiro de 2019: PowerShell Core 6.2

- Última versão da série 6.x
- Foco em estabilidade e compatibilidade

Março de 2020: PowerShell 7.0

- Remoção do termo "Core" do nome
- Baseado em .NET Core 3.1
- **ForEach-Object -Parallel**: Processamento paralelo nativo
- **Operadores ternários**: Sintaxe condicional simplificada
- **Pipeline chain operators** (&&, ||)
- **Null coalescing operators** (??, ??=)
- Compatibilidade aprimorada com Windows PowerShell 5.1

Novembro de 2020: PowerShell 7.1 (LTS)

- Primeira versão Long-Term Support (LTS) do PowerShell 7
- Baseado em .NET 5.0
- Suporte estendido por 3 anos

Novembro de 2021: PowerShell 7.2 (LTS)

- Baseado em .NET 6.0
- Melhorias de performance significativas
- Previsão de suporte até 2024

Novembro de 2022: PowerShell 7.3

- Baseado em .NET 7.0
- Melhorias na experiência do usuário e performance
- Recursos experimentais aprimorados

Novembro de 2023: PowerShell 7.4 (LTS)

- Baseado em .NET 8.0
- Melhorias na segurança e conformidade
- Integração aprimorada com Azure e serviços em nuvem

2024-2025: PowerShell 7.5 (Preview/Atual)

- Evolução contínua com feedback da comunidade
- Foco em performance, segurança e integração com DevOps

1.1.3 Posicionamento no Ecossistema Microsoft

O PowerShell ocupa uma posição estratégica e central no ecossistema Microsoft moderno:

1. Administração de Sistemas Operacionais

- **Windows Server**: Única interface consistente para gerenciar todos os recursos
- **Windows 10/11**: Ferramentas avançadas de administração local
- **Server Core e Nano Server**: Instalações sem GUI dependem exclusivamente do PowerShell

2. Integração com Produtos Microsoft

- **Active Directory**: Módulo ActiveDirectory com centenas de cmdlets
- **Exchange Server**: Toda a administração do Exchange é baseada em PowerShell
- **SharePoint**: Automação e gerenciamento de farms SharePoint

- **SQL Server**: Módulo SQLPS/SqlServer para administração de bancos de dados
- **System Center**: Configuration Manager, Operations Manager, Orchestrator
- **Hyper-V**: Gerenciamento completo de virtualização
- **Azure**: Módulos Az para gerenciar recursos na nuvem
- **Microsoft 365**: Módulos para Exchange Online, SharePoint Online, Teams, etc.

3. DevOps e Automação

- **Azure DevOps**: Tarefas nativas de PowerShell em pipelines
- **GitHub Actions**: Suporte para execução de scripts PowerShell
- **Infrastructure as Code (IaC)**: Scripts de provisionamento e configuração
- **Desired State Configuration (DSC)**: Gerenciamento declarativo de infraestrutura

4. Segurança e Conformidade

- **Microsoft Defender**: Cmdlets para gerenciamento de segurança
- **Antimalware Scan Interface (AMSI)**: Integração nativa
- **JEA (Just Enough Administration)**: Delegação segura de privilégios administrativos
- **Logging e auditoria**: Transcripts, module logging, script block logging

5. Desenvolvimento e Testes

- **Pester**: Framework oficial de testes para PowerShell
- **PSScriptAnalyzer**: Análise estática de código
- **Platyps**: Geração de documentação

1.1.4 Windows PowerShell vs PowerShell 7+

É fundamental compreender as diferenças entre as duas versões:

Aspecto	Windows PowerShell 5.1	PowerShell 7+
Base tecnológica	.NET Framework 4.x	.NET Core / .NET 6+
Plataformas	Apenas Windows	Windows, Linux, macOS

Aspecto	Windows PowerShell 5.1	PowerShell 7+
Ciclo de vida	Manutenção (sem novos recursos)	Desenvolvimento ativo
Instalação no Windows	Nativo (não removível)	Instalação lado a lado
Compatibilidade	100% com módulos legados	~90% compatibilidade
Performance	Boa	Excelente (especialmente I/O)
Módulos exclusivos	Alguns módulos Windows legados	Novos recursos modernos

Quando usar Windows PowerShell 5.1:

- Scripts legados que dependem de módulos incompatíveis
- Ambiente exclusivamente Windows sem necessidade de novos recursos
- Dependências de .NET Framework específicas

Quando usar PowerShell 7+:

- Novos projetos e scripts
- Ambientes multiplataforma
- Necessidade de performance superior
- Uso de recursos modernos da linguagem
- Ambientes em nuvem e containers

1.1.5 Princípios de Design e Filosofia

O PowerShell foi projetado seguindo princípios fundamentais:

1. Orientação a Objetos

- Diferente de shells Unix que trabalham com texto, PowerShell manipula objetos .NET
- Acesso direto a propriedades e métodos
- Eliminação de parsing de texto

2. Consistência

- Convenção de nomenclatura Verbo-Substantivo (Get-Process, Set-Location)
- Parâmetros padronizados em todos os cmdlets
- Comportamento previsível

3. Descoberta

- Sistema de ajuda integrado (Get-Help)
- Autocompletar inteligente (Tab completion)
- Get-Command para descobrir cmdlets

4. Composição

- Pipeline poderoso para encadear comandos
- Cmdlets especializados que fazem uma coisa bem feita
- Filosofia Unix aplicada a objetos

5. Expansibilidade

- Fácil criação de funções e módulos
- Integração com .NET
- Suporte a providers para acessar datastores como filesystem

1.2 Instalação e Configuração do Ambiente

1.2.1 Verificando Versões Instaladas

Antes de instalar ou atualizar o PowerShell, é importante verificar as versões já presentes no sistema.

Verificar versão do Windows PowerShell:

$PSVersionTable

Este comando retorna um objeto hashtable com informações detalhadas:

Name Value

---- -----

PSVersion 5.1.19041.4522

PSEdition Desktop

PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}

BuildVersion 10.0.19041.4522

CLRVersion 4.0.30319.42000

WSManStackVersion 3.0

PSRemotingProtocolVersion 2.3

SerializationVersion 1.1.0.1

Principais propriedades:

- **PSVersion**: Versão do PowerShell instalada
- **PSEdition**: "Desktop" (Windows PowerShell) ou "Core" (PowerShell 7+)
- **CLRVersion**: Versão do Common Language Runtime (.NET)
- **PSCompatibleVersions**: Versões com as quais é compatível

Verificar se PowerShell 7+ está instalado:

 # Verificar via executável

pwsh -v

 # Ou dentro de uma sessão PowerShell 7

$PSVersionTable.PSVersion

1.2.2 Instalando PowerShell 7+ no Windows

Existem várias formas de instalar o PowerShell 7+ no Windows:

Método 1: Via MSI (Instalador tradicional)

1. Acesse o repositório oficial: https://github.com/PowerShell/PowerShell
2. Navegue até a seção "Releases"
3. Baixe o instalador MSI apropriado:
    - PowerShell-7.x.x-win-x64.msi (64 bits)
    - PowerShell-7.x.x-win-x86.msi (32 bits)
4. Execute o instalador e siga o assistente

Opções de instalação:

- Adicionar ao PATH do sistema
- Registrar contexto de menu do Windows Explorer

- Habilitar PowerShell Remoting (requer confirmação)

Método 2: Via Windows Package Manager (winget)

 # Instalar versão estável mais recente

winget install Microsoft.PowerShell

 # Instalar versão Preview

winget install Microsoft.PowerShell.Preview

 # Atualizar PowerShell existente

winget upgrade Microsoft.PowerShell

Método 3: Via Microsoft Store

1. Abra a Microsoft Store
2. Pesquise por "PowerShell"
3. Clique em "Obter" ou "Instalar"
4. Atualizações automáticas são gerenciadas pela Store

Método 4: Via Chocolatey

 # Instalar Chocolatey primeiro (se não tiver)

Set-ExecutionPolicy Bypass -Scope Process -Force

[System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072

iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

 # Instalar PowerShell

choco install powershell-core

Método 5: Via Script de Instalação Automatizada

 # Executar como Administrador

iex "& { $(irm https://aka.ms/install-powershell.ps1) } -UseMSI"

 # Ou com opções específicas

iex "& { $(irm https://aka.ms/install-powershell.ps1) } -UseMSI -Preview -Quiet"

1.2.3 Instalando PowerShell 7+ no Linux

Ubuntu/Debian:

 # Atualizar índice de pacotes

sudo apt-get update

 # Instalar dependências

sudo apt-get install -y wget apt-transport-https software-properties-common

 # Baixar a chave GPG da Microsoft

wget -q "https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/packages-microsoft-prod.deb"

 # Registrar o repositório da Microsoft

sudo dpkg -i packages-microsoft-prod.deb

 # Atualizar lista de pacotes

sudo apt-get update

 # Instalar PowerShell

sudo apt-get install -y powershell

 # Iniciar PowerShell

pwsh

Red Hat Enterprise Linux / CentOS:

 # Registrar repositório da Microsoft

curl https://packages.microsoft.com/config/rhel/8/prod.repo | sudo tee /etc/yum.repos.d/microsoft.repo

 # Instalar PowerShell

sudo yum install -y powershell

 # Iniciar PowerShell

pwsh

Fedora:

 # Importar chave GPG

sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc

 # Registrar repositório

curl https://packages.microsoft.com/config/rhel/8/prod.repo | sudo tee /etc/yum.repos.d/microsoft.repo

 # Instalar PowerShell

sudo dnf install -y powershell

 # Iniciar PowerShell

pwsh

1.2.4 Instalando PowerShell 7+ no macOS

Via Homebrew (recomendado):

 # Instalar Homebrew (se não tiver)

/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

 # Instalar PowerShell

brew install --cask powershell

 # Iniciar PowerShell

pwsh

Via Download Direto:

1. Baixe o pacote PKG de: https://github.com/PowerShell/PowerShell/releases
2. Execute o instalador powershell-7.x.x-osx-x64.pkg
3. Siga o assistente de instalação

1.2.5 Configuração de Execution Policy

O Execution Policy é um mecanismo de segurança que controla a execução de scripts no PowerShell.

Níveis de Execution Policy:

1. **Restricted** (Padrão no Windows cliente)
    - Não permite execução de nenhum script
    - Apenas comandos interativos
2. **AllSigned**
    - Apenas scripts assinados por um publisher confiável
    - Requer certificado digital
3. **RemoteSigned** (Recomendado para desenvolvimento)
    - Scripts locais executam sem assinatura
    - Scripts baixados da internet requerem assinatura
4. **Unrestricted**
    - Executa todos os scripts
    - Avisa sobre scripts da internet
5. **Bypass**

- Nenhuma verificação ou aviso
- Usado em scripts de automação

6. **Undefined**
    - Remove a policy do escopo atual
    - Herda do escopo superior

Escopos de Execution Policy:

 # MachinePolicy: Definido por Group Policy (domínio)

 # UserPolicy: Definido por Group Policy (usuário)

 # Process: Apenas para a sessão atual

 # CurrentUser: Para o usuário atual

 # LocalMachine: Para todos os usuários da máquina

Comandos para gerenciar Execution Policy:

 # Verificar policy atual

Get-ExecutionPolicy

 # Verificar todas as policies por escopo

Get-ExecutionPolicy -List

 # Definir policy para usuário atual (recomendado)

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

 # Definir policy para máquina (requer admin)

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine

 # Definir apenas para sessão atual (não persiste)

Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process

 # Forçar mudança sem confirmação

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Force

Exemplo prático:

 # Situação: você baixou um script da internet

 # O arquivo está bloqueado (Zone.Identifier)

 # Verificar se arquivo está bloqueado

Get-Item .\MeuScript.ps1 -Stream Zone.Identifier

 # Desbloquear arquivo específico

Unblock-File -Path .\MeuScript.ps1

 # Desbloquear todos os scripts de uma pasta

Get-ChildItem -Path C:\Scripts -Recurse | Unblock-File

1.2.6 Configurando o Profile do PowerShell

O profile é um script executado automaticamente ao iniciar o PowerShell, permitindo personalização do ambiente.

Tipos de Profile:

Existem 4 profiles, carregados nesta ordem:

1. **AllUsersAllHosts**: Para todos os usuários, todas as aplicações
    - Caminho: $PSHOME\Profile.ps1
2. **AllUsersCurrentHost**: Para todos os usuários, aplicação atual
    - Caminho: $PSHOME\Microsoft.PowerShell_profile.ps1
3. **CurrentUserAllHosts**: Usuário atual, todas as aplicações
    - Caminho: $Home\Documents\PowerShell\Profile.ps1
4. **CurrentUserCurrentHost**: Usuário atual, aplicação atual (mais comum)
    - Caminho: $Home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1

Verificando caminhos dos profiles:

 # Ver todos os profiles disponíveis

```
$PROFILE | Get-Member -Type NoteProperty | Select-Object Name,
@{Name='Path'; Expression={$PROFILE.$($_.Name)}}

# Ver caminho do profile atual
$PROFILE

# Verificar se profile existe
Test-Path $PROFILE
```

Criando um profile:

```
# Criar diretório se não existir
if (!(Test-Path (Split-Path $PROFILE))) {
    New-Item -Path (Split-Path $PROFILE) -ItemType Directory -Force
}

# Criar arquivo de profile
New-Item -Path $PROFILE -ItemType File -Force

# Abrir no editor padrão
notepad $PROFILE

# Ou no VS Code
code $PROFILE
```

Exemplo de Profile Básico:

```
# Profile para PowerShell 7
# Localização: $Home\Documents\PowerShell\Microsoft.PowerShell_profile.ps1

# Mensagem de boas-vindas
Write-Host "Bem-vindo, $env:USERNAME!" -ForegroundColor Green
```

```
Write-Host "PowerShell $($PSVersionTable.PSVersion)" -ForegroundColor Cyan

# Aliases personalizados
Set-Alias -Name np -Value notepad.exe
Set-Alias -Name ll -Value Get-ChildItem

# Funções personalizadas
function prompt {
  $location = Get-Location
  Write-Host "PS " -NoNewline -ForegroundColor Yellow
  Write-Host "$location" -NoNewline -ForegroundColor Cyan
  Write-Host ">" -NoNewline -ForegroundColor Yellow
  return " "
}

function Get-MyIP {
  (Invoke-WebRequest -Uri 'https://api.ipify.org').Content
}

# Importar módulos frequentemente usados
Import-Module -Name PSReadLine

# Configurações do PSReadLine
Set-PSReadLineOption -PredictionSource History
Set-PSReadLineOption -PredictionViewStyle ListView
Set-PSReadLineOption -EditMode Windows
Set-PSReadLineKeyHandler -Key Tab -Function MenuComplete
```

```
# Variáveis de ambiente personalizadas
$env:SCRIPTS_PATH = "C:\Scripts"

# Navegação rápida
function GoScripts { Set-Location $env:SCRIPTS_PATH }
Set-Alias -Name scripts -Value GoScripts

# Histórico aumentado
$MaximumHistoryCount = 10000
```

1.2.7 Instalando e Configurando Ferramentas Complementares

Visual Studio Code com PowerShell Extension

Visual Studio Code é o editor recomendado para desenvolvimento em PowerShell.

```
# Instalar VS Code via winget
winget install Microsoft.VisualStudioCode

# Instalar extensão PowerShell
code --install-extension ms-vscode.powershell
```

Configurações recomendadas para VS Code (settings.json):

```
{
  "powershell.codeFormatting.preset": "OTBS",
  "powershell.codeFormatting.autoCorrectAliases": true,
  "powershell.codeFormatting.useCorrectCasing": true,
  "powershell.integratedConsole.focusConsoleOnExecute": false,
  "powershell.scriptAnalysis.enable": true,
  "editor.formatOnSave": true,
  "files.trimTrailingWhitespace": true
}
```

Windows Terminal

Windows Terminal oferece uma experiência moderna de linha de comando:

 # Instalar via winget

winget install Microsoft.WindowsTerminal

 # Ou via Microsoft Store

 # Pesquisar: "Windows Terminal"

Configuração de perfil padrão (settings.json):

```
{
  "defaultProfile": "{574e775e-4f2a-5b96-ac1e-a2962a402336}",
  "profiles": {
    "defaults": {
      "fontFace": "Cascadia Code",
      "fontSize": 11,
      "colorScheme": "Campbell Powershell"
    },
    "list": [
      {
        "guid": "{574e775e-4f2a-5b96-ac1e-a2962a402336}",
        "name": "PowerShell 7",
        "source": "Windows.Terminal.PowershellCore",
        "icon": "ms-appx:///ProfileIcons/{574e775e-4f2a-5b96-ac1e-a2962a402336}.png",
        "startingDirectory": "%USERPROFILE%"
      }
    ]
  }
```

}

Oh My Posh (Temas e personalização do prompt)

 # Instalar Oh My Posh

winget install JanDeDobbeleer.OhMyPosh

 # Instalar fontes Nerd Fonts

oh-my-posh font install

 # Adicionar ao profile

oh-my-posh init pwsh | Invoke-Expression

 # Ou com tema específico

oh-my-posh init pwsh --config "$env:POSH_THEMES_PATH/paradox.omp.json" | Invoke-Expression

PSReadLine (Melhorias na linha de comando)

PSReadLine já vem instalado no PowerShell 7, mas pode ser atualizado:

 # Atualizar PSReadLine

Install-Module -Name PSReadLine -Force -AllowClobber

 # Configurações recomendadas (adicionar ao profile)

Set-PSReadLineOption -PredictionSource History

Set-PSReadLineOption -PredictionViewStyle ListView

Set-PSReadLineOption -HistorySearchCursorMovesToEnd

Set-PSReadLineKeyHandler -Key UpArrow -Function HistorySearchBackward

Set-PSReadLineKeyHandler -Key DownArrow -Function HistorySearchForward

1.2.8 Configurando PowerShell Remoting

PowerShell Remoting permite executar comandos em computadores remotos.

Habilitar Remoting (no computador de destino):

 # Executar como Administrador

Enable-PSRemoting -Force

 # Verificar configuração

Get-PSSessionConfiguration

 # Testar conectividade local

Test-WSMan

Configurações de firewall (automáticas com Enable-PSRemoting):

- Porta 5985 (HTTP - WinRM)
- Porta 5986 (HTTPS - WinRM-HTTPS)

Configurar clientes confiáveis (se não estiver em domínio):

 # No computador cliente, executar como Administrador

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "192.168.1.100,Server01" -Force

 # Ou permitir todos (não recomendado em produção)

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "*" -Force

 # Verificar configuração

Get-Item WSMan:\localhost\Client\TrustedHosts

Testar conexão remota:

 # Teste básico

Test-NetConnection -ComputerName Server01 -Port 5985

 # Teste de sessão remota

Enter-PSSession -ComputerName Server01 -Credential (Get-Credential)

1.2.9 Módulos Essenciais para Instalar

PowerShellGet (gerenciador de módulos):

 # Verificar versão

Get-Module -Name PowerShellGet -ListAvailable

 # Atualizar para versão mais recente

Install-Module -Name PowerShellGet -Force -AllowClobber

 # Registrar PSGallery como repositório confiável

Set-PSRepository -Name PSGallery -InstallationPolicy Trusted

Módulos recomendados:

 # Pester - Framework de testes

Install-Module -Name Pester -Force

 # PSScriptAnalyzer - Análise estática de código

Install-Module -Name PSScriptAnalyzer -Force

 # ImportExcel - Manipulação de arquivos Excel sem Microsoft Office

Install-Module -Name ImportExcel -Force

 # Az - Gerenciamento do Azure

Install-Module -Name Az -AllowClobber -Force

 # Microsoft.Graph - Gerenciamento do Microsoft 365

Install-Module -Name Microsoft.Graph -Force

 # SqlServer - Administração do SQL Server

Install-Module -Name SqlServer -Force

Verificar módulos instalados:

 # Listar todos os módulos disponíveis

Get-Module -ListAvailable

 # Buscar módulos na galeria

Find-Module -Name *Azure*

 # Ver informações de um módulo

Get-Module -Name Az -ListAvailable | Select-Object Name, Version, Description

1.2.10 Verificação Final da Instalação

Execute este script completo para validar sua instalação:

 # Script de Verificação do Ambiente PowerShell

Write-Host "` n=== VERIFICAÇÃO DO AMBIENTE POWERSHELL ===" -ForegroundColor Cyan

 # Versão do PowerShell

Write-Host "` n[1] Versão do PowerShell:" -ForegroundColor Yellow

$PSVersionTable | Format-Table -AutoSize

 # Execution Policy

Write-Host "` n[2] Execution Policy:" -ForegroundColor Yellow

Get-ExecutionPolicy -List | Format-Table -AutoSize

 # Profile

Write-Host "` n[3] Profile do PowerShell:" -ForegroundColor Yellow

Write-Host "Caminho: $PROFILE"

Write-Host "Existe: $(Test-Path $PROFILE)"

 # Módulos Importantes

```
Write-Host "`n[4] Módulos Essenciais:" -ForegroundColor Yellow
$modulos = @('PowerShellGet', 'PSReadLine', 'Pester', 'PSScriptAnalyzer')
foreach ($mod in $modulos) {
  $versao = (Get-Module -Name $mod -ListAvailable | Select-Object -First 1).Version
  if ($versao) {
    Write-Host "$mod : $versao" -ForegroundColor Green
  } else {
    Write-Host "$mod : NÃO INSTALADO" -ForegroundColor Red
  }
}

# PowerShell Remoting
Write-Host "`n[5] PowerShell Remoting:" -ForegroundColor Yellow
try {
  Test-WSMan -ErrorAction Stop | Out-Null
  Write-Host "Status: HABILITADO" -ForegroundColor Green
} catch {
  Write-Host "Status: DESABILITADO" -ForegroundColor Red
}

# Repositórios
Write-Host "`n[6] Repositórios Configurados:" -ForegroundColor Yellow
Get-PSRepository | Format-Table -AutoSize

Write-Host "`n=== VERIFICAÇÃO CONCLUÍDA ===" -ForegroundColor Cyan
```

Conclusão da Seção 1

Nesta primeira seção, exploramos em profundidade a história e evolução do PowerShell, desde sua concepção como Projeto Monad até as versões modernas

multiplataforma. Compreendemos seu posicionamento estratégico no ecossistema Microsoft e as diferenças fundamentais entre Windows PowerShell 5.1 e PowerShell 7+.

Também realizamos uma configuração completa do ambiente, incluindo instalação em diferentes sistemas operacionais, configuração de execution policies, criação de profiles personalizados e instalação de ferramentas complementares essenciais.

Com este ambiente preparado, você está pronto para começar a explorar os recursos e capacidades do PowerShell de forma prática e eficiente.

2. COMANDOS E SINTAXE BÁSICA

2.1 Cmdlets, Parâmetros e Pipeline

2.1.1 Conceito e Estrutura de Cmdlets

O que é um Cmdlet?

Um **cmdlet** (pronuncia-se "command-let") é a unidade fundamental de comando no PowerShell. Diferente de comandos tradicionais que são programas executáveis, cmdlets são classes .NET especializadas compiladas em assemblies que são carregados pelo PowerShell.

Características fundamentais dos cmdlets:

- São comandos nativos do PowerShell escritos em C# ou PowerShell
- Seguem um padrão consistente de nomenclatura
- Trabalham com objetos .NET, não apenas texto
- Suportam parâmetros padronizados
- Integram-se perfeitamente ao pipeline
- Possuem sistema de ajuda incorporado

Convenção de Nomenclatura: Verbo-Substantivo

Todos os cmdlets seguem a convenção **Verbo-Substantivo** (Verb-Noun), tornando-os intuitivos e fáceis de lembrar.

Estrutura:

Verbo-Substantivo

| |

│ └── O que está sendo manipulado (objeto)

└──────── Ação que está sendo executada

Exemplos práticos:

Get-Process *# Obtém processos em execução*

Stop-Service *# Para um serviço*

New-Item *# Cria um novo item (arquivo, pasta, etc.)*

Set-Location *# Define a localização atual*

Remove-Item *# Remove um item*

Test-Connection *# Testa conectividade de rede*

Verbos Aprovados

O PowerShell possui uma lista de verbos aprovados para garantir consistência. Os verbos são agrupados por categoria:

Verbos Comuns (Common):

Get-Verb | Where-Object Group -eq 'Common'

Verbo	**Significado**	**Exemplo**
Get	Obtém um recurso	Get-Service
Set	Define ou modifica um recurso	Set-ExecutionPolicy
New	Cria um novo recurso	New-Item
Remove	Exclui um recurso	Remove-Item
Add	Adiciona a uma coleção	Add-Content
Clear	Remove conteúdo mas mantém o objeto	Clear-Content
Copy	Copia um recurso	Copy-Item
Move	Move um recurso	Move-Item
Rename	Renomeia um recurso	Rename-Item

Verbos de Dados (Data):

Verbo	Significado	Exemplo
Import	Importa dados	Import-Csv
Export	Exporta dados	Export-Clixml
ConvertTo	Converte para formato	ConvertTo-Json
ConvertFrom	Converte de formato	ConvertFrom-Json
Compare	Compara objetos	Compare-Object
Select	Seleciona propriedades	Select-Object

Verbos de Comunicação (Communications):

Verbo	Significado	Exemplo
Connect	Estabelece conexão	Connect-PSSession
Disconnect	Encerra conexão	Disconnect-PSSession
Read	Lê dados	Read-Host
Write	Escreve dados	Write-Output
Send	Envia dados	Send-MailMessage
Receive	Recebe dados	Receive-Job

Verbos de Ciclo de Vida (Lifecycle):

Verbo	Significado	Exemplo
Start	Inicia um recurso	Start-Service
Stop	Para um recurso	Stop-Process

Verbo	Significado	Exemplo
Restart	Reinicia um recurso	Restart-Computer
Suspend	Pausa um recurso	Suspend-Service
Resume	Retoma um recurso	Resume-Service

Listar todos os verbos aprovados:

```
# Ver todos os verbos
Get-Verb | Format-Table -AutoSize

# Contar verbos por grupo
Get-Verb | Group-Object Group | Sort-Object Count -Descending

# Buscar verbos específicos
Get-Verb -Verb *move*
```

Anatomia de um Cmdlet

Vamos analisar a estrutura completa de um cmdlet:

```
Get-ChildItem -Path C:\Users -Filter *.txt -Recurse -Force
│           │     │          │     │
│           │     │          │     └─ Parâmetro switch
│           │     │          └─────── Parâmetro switch
│           │     └────────────────── Parâmetro nomeado
│           └──────────────────────── Parâmetro nomeado
└──────────────────────────────────── Cmdlet
```

Componentes:

1. **Cmdlet**: O comando principal (Get-ChildItem)
2. **Parâmetros nomeados**: Aceitam valores (-Path C:\Users)

3. **Parâmetros switch**: Booleanos, ativados pela presença (-Recurse)
4. **Valores**: Dados passados aos parâmetros (C:\Users, *.txt)

2.1.2 Parâmetros

Tipos de Parâmetros

1. Parâmetros Posicionais

Podem ser especificados sem o nome, baseados na posição:

```
# Com nome do parâmetro
Get-ChildItem -Path C:\Windows

# Sem nome (posicional) - funciona se Path for o primeiro parâmetro
Get-ChildItem C:\Windows

# Múltiplos parâmetros posicionais
Copy-Item C:\origem.txt C:\destino.txt
# Equivalente a:
Copy-Item -Path C:\origem.txt -Destination C:\destino.txt
```

2. Parâmetros Nomeados

Especificados explicitamente com o nome:

```
Get-Service -Name wuauserv
Get-Process -Name notepad
Get-Content -Path arquivo.txt -Encoding UTF8
```

Vantagens dos parâmetros nomeados:

- Clareza e legibilidade
- Não dependem de ordem
- Autocompletar funciona melhor
- Recomendados em scripts de produção

3. Parâmetros Switch

Parâmetros booleanos que não requerem valor:

 # Ativar o switch

Get-ChildItem -Recurse

Get-Process -IncludeUserName

 # Desativar explicitamente (raro)

Get-ChildItem -Recurse:$false

 # Atribuir a variável

$recursivo = $true

Get-ChildItem -Recurse:$recursivo

4. Parâmetros Obrigatórios vs Opcionais

 # Obrigatório - cmdlet solicita se não for fornecido

New-Item -ItemType File

 # PowerShell perguntará: Path?

 # Opcional - tem valor padrão

Get-ChildItem

 # Usa o diretório atual como padrão

Parâmetros Comuns (Common Parameters)

Todos os cmdlets avançados suportam parâmetros comuns automaticamente:

 # -Verbose: Mostra mensagens detalhadas

Get-Process -Verbose

 # -Debug: Mostra mensagens de depuração

Get-Service -Debug

 # -ErrorAction: Controla comportamento de erros

Get-Item "C:\naoexiste.txt" -ErrorAction SilentlyContinue

 # -ErrorVariable: Armazena erros em variável

Get-Service -Name "ServicoInexistente" -ErrorVariable meuErro

$meuErro

 # -WarningAction: Controla exibição de avisos

Get-Process -WarningAction Ignore

 # -InformationAction: Controla mensagens informativas

Write-Information "Processando..." -InformationAction Continue

 # -OutVariable: Captura saída em variável

Get-Process -Name notepad -OutVariable processos

$processos

 # -WhatIf: Simula execução sem executar (dry-run)

Remove-Item C:\teste.txt -WhatIf

 # -Confirm: Solicita confirmação

Stop-Service wuauserv -Confirm

ErrorAction - Valores possíveis:

 # Continue (padrão): Exibe erro e continua

Get-Item "C:\naoexiste.txt" -ErrorAction Continue

 # SilentlyContinue: Suprime erro e continua

Get-Item "C:\naoexiste.txt" -ErrorAction SilentlyContinue

 # Stop: Trata como erro terminante

Get-Item "C:\naoexiste.txt" -ErrorAction Stop

 # Inquire: Pergunta ao usuário o que fazer

Get-Item "C:\naoexiste.txt" -ErrorAction Inquire

 # Ignore: Ignora completamente (não adiciona ao $Error)

Get-Item "C:\naoexiste.txt" -ErrorAction Ignore

Descobrindo Parâmetros

Usando Get-Command:

 # Ver sintaxe básica

Get-Command Get-Process

 # Ver todos os conjuntos de parâmetros

Get-Command Get-Process -Syntax

 # Detalhes completos

Get-Command Get-Process | Format-List *

Usando Get-Help:

 # Ajuda completa

Get-Help Get-Process -Full

 # Apenas parâmetros

Get-Help Get-Process -Parameter *

 # Parâmetro específico

Get-Help Get-Process -Parameter Name

 # Exemplos práticos

Get-Help Get-Process -Examples

Usando IntelliSense e Tab Completion:

 # Digite o cmdlet e pressione espaço, depois Ctrl+Espaço

Get-Process <Ctrl+Espaço>

 # Ou use Tab para percorrer parâmetros

Get-Process -<Tab>

 # Tab completion também funciona com valores

Get-Service -Name w<Tab>

Abreviação de Parâmetros

O PowerShell permite abreviar nomes de parâmetros, desde que a abreviação seja única:

 # Completo

Get-Process -Name notepad

 # Abreviado

Get-Process -N notepad

 # Também funciona

Get-Process -Na notepad

 # Erro: ambíguo (pode ser Name ou Path)

Get-ChildItem -P *# ❌ Erro*

⚠️ **Aviso importante:** Embora abreviações funcionem, **NÃO as use em scripts de produção**. Use sempre nomes completos para clareza e manutenibilidade.

Passando Múltiplos Valores

Muitos parâmetros aceitam arrays:

 # Array explícito

Get-Process -Name notepad, chrome, explorer

 # Array criado com operador de vírgula

$processos = "notepad", "chrome"

Get-Process -Name $processos

 # Pipeline

"notepad", "chrome", "explorer" | Get-Process

 # Range (apenas para tipos apropriados)

1..10 | Get-Random -Count 3

2.1.3 Pipeline - O Coração do PowerShell

Conceito e Funcionamento

O **pipeline** (representado pelo símbolo |) é um dos recursos mais poderosos do PowerShell. Ele permite encadear comandos, passando a saída de um como entrada do próximo.

Diferença fundamental em relação a shells Unix:

 # Unix/Linux (passa TEXTO)

ls -l | grep ".txt"

 # PowerShell (passa OBJETOS)

Get-ChildItem | Where-Object Extension -eq ".txt"

Fluxo de dados no pipeline:

Cmdlet 1 → Objeto → Cmdlet 2 → Objeto → Cmdlet 3 → Resultado

Pipeline Básico

 # Obter processos e filtrar

Get-Process | Where-Object CPU -gt 100

 # Obter serviços, filtrar e ordenar

Get-Service | Where-Object Status -eq 'Running' | Sort-Object Name

 # Obter arquivos, selecionar propriedades

Get-ChildItem | Select-Object Name, Length, LastWriteTime

 # Cadeia longa

Get-Process |

  Where-Object WorkingSet -gt 100MB |

  Sort-Object WorkingSet -Descending |

  Select-Object -First 10 |

  Format-Table Name, Id, @{Name='RAM(MB)';Expression={$_.WorkingSet / 1MB}} -AutoSize

Vinculação de Pipeline (Pipeline Binding)

O PowerShell usa dois métodos para vincular objetos no pipeline aos parâmetros:

1. ByValue (Por Valor)

O objeto inteiro é passado e o cmdlet procura um parâmetro que aceite esse tipo:

 # String é passada para o parâmetro -Name

"wuauserv", "spooler" | Get-Service

 # Como descobrir o que aceita ByValue

Get-Help Get-Service -Parameter Name

 # Verá: Accept pipeline input: True (ByValue, ByPropertyName)

2. ByPropertyName (Por Nome de Propriedade)

Propriedades do objeto são mapeadas para parâmetros com o mesmo nome:

 # Objeto tem propriedade 'Name', mapeada para -Name

Get-Service wuauserv | Stop-Service

 # Verificar mapeamento

Get-Service wuauserv | Get-Member -MemberType Properties

 # Verá a propriedade Name

Get-Help Stop-Service -Parameter Name

 # Verá: Accept pipeline input: True (ByPropertyName, ByValue)

Exemplo complexo de ByPropertyName:

 # Criar objetos customizados

```
$computadores = @(
  [PSCustomObject]@{ComputerName = 'Server01'; Path = 'C:\Logs'}
  [PSCustomObject]@{ComputerName = 'Server02'; Path = 'C:\Logs'}
)

# As propriedades são automaticamente mapeadas
$computadores | Get-ChildItem
# ComputerName → -ComputerName
# Path → -Path
```

Pipeline Variables (Variáveis Automáticas)

Dentro do pipeline, existem variáveis especiais:

```
# $_ ou $PSItem - representa o objeto atual
Get-Process | Where-Object { $_.CPU -gt 100 }

# Acessando propriedades
Get-Service | ForEach-Object { "Serviço: $($_.Name) - Status: $($_.Status)" }

# Chamando métodos
Get-Process notepad | ForEach-Object { $_.Kill() }
```

Cmdlets Essenciais para Pipeline

Where-Object (Filtragem):

```
# Sintaxe simplificada (PS 3.0+)
Get-Process | Where-Object CPU -gt 50

# Sintaxe de script block (mais flexível)
Get-Process | Where-Object { $_.CPU -gt 50 -and $_.WorkingSet -gt 100MB }

# Múltiplas condições
Get-Service | Where-Object {
  $_.Status -eq 'Running' -and
  $_.StartType -eq 'Automatic'
}

# Operadores comuns
Get-ChildItem | Where-Object Name -like "*.txt"
Get-ChildItem | Where-Object Length -ge 1MB
Get-Process | Where-Object ProcessName -match "^chrome"
```

Select-Object (Seleção e Transformação):

```
# Selecionar propriedades específicas
Get-Process | Select-Object Name, Id, CPU

# Primeiros N elementos
Get-Process | Select-Object -First 5

# Últimos N elementos
Get-Process | Select-Object -Last 5

# Pular elementos
```

```
Get-Process | Select-Object -Skip 10 -First 5

# Propriedades únicas
Get-Process | Select-Object ProcessName -Unique

# Propriedades calculadas
Get-Process | Select-Object Name,
  @{Name='CPUTime'; Expression={$_.CPU}},
  @{Name='MemoryMB'; Expression={$_.WorkingSet / 1MB}}

# Propriedades calculadas com formatação
Get-ChildItem | Select-Object Name,
  @{Name='SizeMB'; Expression={'{0:N2}' -f ($_.Length / 1MB)}}
```

Sort-Object (Ordenação):

```
# Ordenação ascendente (padrão)
Get-Process | Sort-Object CPU

# Ordenação descendente
Get-Process | Sort-Object CPU -Descending

# Múltiplas chaves
Get-Process | Sort-Object Company, Name

# Com tipos diferentes
Get-ChildItem | Sort-Object LastWriteTime -Descending

# Ordenação customizada (caso sensível)
Get-Service | Sort-Object Name -CaseSensitive
```

ForEach-Object (Iteração):

```
# Sintaxe básica
Get-Process | ForEach-Object { $_.Name }

# Múltiplas operações
Get-ChildItem -File | ForEach-Object {
    Write-Host "Processando: $($_.Name)"
    $_.LastWriteTime = Get-Date
}

# Com Begin, Process, End
1..10 | ForEach-Object -Begin {
    Write-Host "Iniciando..."
    $soma = 0
} -Process {
    $soma += $_
} -End {
    Write-Host "Soma total: $soma"
}

# Processamento paralelo (PS 7.0+)
1..10 | ForEach-Object -Parallel {
    Start-Sleep -Seconds 1
    "Processado: $_"
} -ThrottleLimit 5
```

Group-Object (Agrupamento):

```
# Agrupar por propriedade
Get-Service | Group-Object Status
```

```
# Com contagem
Get-Process | Group-Object Company | Sort-Object Count -Descending

# Múltiplas propriedades
Get-ChildItem | Group-Object Extension, @{Expression={
  if($_.Length -gt 1MB){'Grande'}else{'Pequeno'}
}}

# Acessar grupos
$grupos = Get-Process | Group-Object Company
$grupos | ForEach-Object {
  Write-Host "$($_.Name): $($_.Count) processos"
}
```

Measure-Object (Cálculos):

```
# Contar objetos
Get-Process | Measure-Object

# Soma
Get-ChildItem -File | Measure-Object -Property Length -Sum

# Estatísticas completas
Get-Process | Measure-Object -Property CPU -Average -Sum -Maximum -Minimum

# Múltiplas propriedades
Get-Process | Measure-Object -Property CPU, WorkingSet -Average
```

Formatação de Saída

⚠️ **IMPORTANTE:** Cmdlets de formatação (Format-*) devem ser **sempre os últimos** no pipeline, pois convertem objetos em formatação, quebrando o pipeline.

Format-Table:

 # Básico

Get-Process | Format-Table

 # Propriedades específicas

Get-Process | Format-Table Name, Id, CPU

 # AutoSize para ajustar colunas

Get-Process | Format-Table Name, Id, CPU -AutoSize

 # GroupBy

Get-Service | Format-Table -GroupBy Status

 # Propriedades calculadas

Get-Process | Format-Table Name,

  @{Label='CPU(s)'; Expression={$_.CPU}; Width=10; Alignment='Right'}

Format-List:

 # Todas as propriedades

Get-Process -Name powershell | Format-List *

 # Propriedades específicas

Get-Service wuauserv | Format-List Name, Status, StartType

 # Útil para objetos complexos

Get-ComputerInfo | Format-List

Format-Wide:

 # Exibição em colunas

Get-Process | Format-Wide Name

 # Especificar número de colunas

Get-Process | Format-Wide Name -Column 4

Out-GridView:

 # Interface gráfica interativa

Get-Process | Out-GridView

 # Com seleção

Get-Service | Out-GridView -PassThru | Start-Service

 # Modo de saída

$selecionados = Get-Process | Out-GridView -OutputMode Multiple

Exportação de Dados

Out-File:

 # Salvar em arquivo texto

Get-Process | Out-File processos.txt

 # Com encoding específico

Get-Process | Out-File processos.txt -Encoding UTF8

 # Append

Get-Service | Out-File servicos.txt -Append

 # Largura de linha

Get-Process | Out-File processos.txt -Width 200

Export-Csv:

 # Exportar para CSV

Get-Process | Export-Csv processos.csv

 # Sem informações de tipo

Get-Process | Export-Csv processos.csv -NoTypeInformation

 # Com delimitador customizado

Get-Process | Export-Csv processos.csv -Delimiter ';'

 # Encoding

Get-Service | Export-Csv servicos.csv -Encoding UTF8

Export-Clixml:

 # Serialização completa de objetos

Get-Process | Export-Clixml processos.xml

 # Importar posteriormente

$processos = Import-Clixml processos.xml

ConvertTo-Json / ConvertTo-Html:

 # JSON

Get-Process | Select-Object Name, Id, CPU | ConvertTo-Json

 # JSON com profundidade

Get-ComputerInfo | ConvertTo-Json -Depth 3

 # HTML

Get-Service | ConvertTo-Html -Title "Serviços" | Out-File servicos.html

 # HTML com CSS

```
$css = @"
<style>
  body { font-family: Arial; }
  table { border-collapse: collapse; }
  th, td { border: 1px solid black; padding: 5px; }
  th { background-color: #4CAF50; color: white; }
</style>
"@
Get-Process | ConvertTo-Html -Head $css | Out-File processos.html
```

Pipeline Avançado - Técnicas

1. Pipeline com múltiplas transformações:

```
Get-EventLog -LogName Application -Newest 100 |
  Where-Object EntryType -eq 'Error' |
  Group-Object Source |
  Select-Object Name, Count |
  Sort-Object Count -Descending |
  Format-Table -AutoSize
```

2. Pipeline com foreach e variáveis:

```
$relatorio = Get-Process |
  Where-Object CPU -gt 10 |
  ForEach-Object {
    [PSCustomObject]@{
      Nome = $_.Name
      CPU = [math]::Round($_.CPU, 2)
      MemoriaMB = [math]::Round($_.WorkingSet / 1MB, 2)
      Threads = $_.Threads.Count
    }
  }
```

```
$relatorio | Export-Csv relatorio_processos.csv -NoTypeInformation
```

3. Pipeline com validação:

```
Get-ChildItem -Path C:\Logs -Filter *.log |
  Where-Object LastWriteTime -lt (Get-Date).AddDays(-30) |
  ForEach-Object {
    Write-Host "Deletando arquivo antigo: $($_.Name)"
    Remove-Item $_.FullName -WhatIf
  }
```

4. Pipeline com tratamento de erros:

```
$servidores = 'Server01', 'Server02', 'ServerInexistente'


$servidores | ForEach-Object {
  try {
    Test-Connection -ComputerName $_ -Count 1 -ErrorAction Stop
    [PSCustomObject]@{
      Servidor = $_
      Status = 'Online'
      Erro = $null
    }
  }
  catch {
    [PSCustomObject]@{
      Servidor = $_
      Status = 'Offline'
      Erro = $_.Exception.Message
    }
  }
```

} | Format-Table -AutoSize

2.1.4 Descoberta de Comandos

Get-Command - Encontrando Cmdlets

```
# Todos os comandos disponíveis
Get-Command

# Filtrar por tipo
Get-Command -CommandType Cmdlet
Get-Command -CommandType Function
Get-Command -CommandType Alias

# Buscar por padrão
Get-Command -Name *Service*
Get-Command -Verb Get
Get-Command -Noun Process

# Buscar em módulo específico
Get-Command -Module Microsoft.PowerShell.Management

# Buscar com wildcard
Get-Command Get-*Item*

# Ver definição
Get-Command Get-Process | Format-List *
```

Get-Help - Sistema de Ajuda

```
# Ajuda básica
Get-Help Get-Process
```

 # Ajuda detalhada

Get-Help Get-Process -Detailed

 # Ajuda completa

Get-Help Get-Process -Full

 # Apenas exemplos

Get-Help Get-Process -Examples

 # Ajuda online (abre no browser)

Get-Help Get-Process -Online

 # Buscar na ajuda

Get-Help *firewall*

 # Atualizar arquivos de ajuda

Update-Help -Force

Get-Member - Explorando Objetos

 # Ver membros de um objeto

Get-Process | Get-Member

 # Apenas propriedades

Get-Process | Get-Member -MemberType Property

 # Apenas métodos

Get-Process | Get-Member -MemberType Method

 # Buscar membro específico

Get-Process | Get-Member -Name *Memory*

 # Ver tipo do objeto

(Get-Process)[0].GetType()

Aliases - Atalhos para Comandos

 # Ver todos os aliases

Get-Alias

 # Alias específico

Get-Alias ls

 # Qual comando por trás do alias

Get-Alias -Definition Get-ChildItem

 # Criar alias temporário

Set-Alias -Name np -Value notepad.exe

 # Criar alias permanente (adicionar ao profile)

New-Alias -Name gh -Value Get-Help

 # Remover alias

Remove-Alias -Name np

Aliases comuns do PowerShell:

Alias	**Cmdlet**	**Origem**
ls, dir, gci	Get-ChildItem	Unix/DOS/PS
cd, chdir, sl	Set-Location	DOS/PS
cp, copy, cpi	Copy-Item	Unix/DOS/PS

Alias	Cmdlet	Origem
mv, move, mi	Move-Item	Unix/DOS/PS
rm, del, erase, ri	Remove-Item	Unix/DOS/PS
cat, type, gc	Get-Content	Unix/DOS/PS
ps, gps	Get-Process	Unix/PS
kill, spps	Stop-Process	Unix/PS
cls, clear	Clear-Host	DOS/Unix
man, help	Get-Help	Unix/DOS
pwd, gl	Get-Location	Unix/PS

⚠️ **Importante:** Evite usar aliases em scripts de produção. Use sempre os nomes completos dos cmdlets para clareza e manutenibilidade.

2.2 Estruturas Condicionais e de Repetição

2.2.1 Variáveis no PowerShell

Antes de trabalhar com estruturas de controle, precisamos entender variáveis.

Declaração e Atribuição

```
# Declaração simples (tipagem dinâmica)
$nome = "João"
$idade = 30
$ativo = $true

# PowerShell infere o tipo automaticamente
$numero = 42        # System.Int32
$texto = "PowerShell" # System.String
$decimal = 3.14     # System.Double
```

 # Tipagem explícita (fortemente tipado)

[int]$quantidade = 100

[string]$mensagem = "Olá"

[datetime]$data = "2025-01-15"

[bool]$habilitado = $true

 # Múltiplas atribuições

$a, $b, $c = 1, 2, 3

$x = $y = $z = 0

Tipos de Dados Comuns

 # Numéricos

[byte]$byte = 255 *# 0-255*

[int16]$curto = 32767 *# -32,768 a 32,767*

[int]$inteiro = 2147483647 *# -2.1B a 2.1B*

[long]$longo = 9223372036854775807

[single]$flutuante = 3.14159

[double]$duplo = 3.141592653589793

[decimal]$monetario = 99.99

 # Texto

[string]$texto = "PowerShell"

[char]$caractere = 'A'

 # Booleano

[bool]$verdadeiro = $true

[bool]$falso = $false

```
# Data e hora
[datetime]$agora = Get-Date
[datetime]$especifica = "2025-10-15 15:00:00"

# Arrays
[array]$numeros = 1, 2, 3, 4, 5
[int[]]$inteiros = 10, 20, 30

# Hashtables
[hashtable]$pessoa = @{
  Nome = "Maria"
  Idade = 28
  Cidade = "São Paulo"
}

# Objetos customizados
[PSCustomObject]$produto = @{
  Nome = "Notebook"
  Preco = 3500.00
  Estoque = 15
}
```

Variáveis Automáticas

PowerShell possui variáveis automáticas predefinidas:

```
# Variáveis do sistema
$PSVersionTable   # Informações da versão
$HOME          # Diretório home do usuário
$PWD           # Diretório atual
$PID           # ID do processo atual
```

```
# Variáveis de resultado
$?          # Status do último comando (true/false)
$_ ou $PSItem  # Item atual no pipeline
$^          # Primeiro token do último comando
$$          # Último token do último comando

# Variáveis de erro
$Error        # Array com todos os erros
$LastExitCode    # Código de saída do último programa nativo

# Outras úteis
$null         # Valor nulo
$true / $false   # Booleanos
```

Escopo de Variáveis

```
# Escopo local (padrão)
$local = "Visível apenas aqui"

# Escopo script (todo o script)
$script:configuracao = "Disponível no script inteiro"

# Escopo global (toda a sessão)
$global:importante = "Visível em todo lugar"

# Escopo privado (não herdado por child scopes)
$private:secreto = "Não vaza"

# Exemplo de uso
```

```
function Teste {
   $local = "Função"
   $script:nivel = "Script"
   Write-Host "Local: $local"
}
Teste
Write-Host "Script: $nivel"
# Write-Host "Local: $local" # ❌ Erro - fora do escopo
```

2.2.2 Operadores

Operadores Aritméticos

```
# Operações básicas
$a = 10
$b = 3

$a + $b   # 13 - Adição
$a - $b   # 7  - Subtração
$a * $b   # 30 - Multiplicação
$a / $b   # 3.333... - Divisão
$a % $b   # 1  - Módulo (resto)

# Incremento e decremento
$contador = 0
$contador++   # 1
$contador--   # 0
++$contador   # 1 (pré-incremento)
--$contador   # 0 (pré-decremento)

# Operações compostas
```

```
$total = 100
$total += 50   # $total = $total + 50
$total -= 20   # $total = $total - 20
$total *= 2    # $total = $total * 2
$total /= 4    # $total = $total / 4
```

Operadores de Comparação

```
# Igualdade
$a -eq $b     # Igual a
$a -ne $b     # Diferente de

# Maior/Menor
$a -gt $b     # Maior que
$a -ge $b     # Maior ou igual
$a -lt $b     # Menor que
$a -le $b     # Menor ou igual

# Strings (case-insensitive por padrão)
"PowerShell" -eq "powershell"    # True
"PowerShell" -ceq "powershell"   # False (case-sensitive)

# Like (wildcard)
"PowerShell" -like "Power*"     # True
"PowerShell" -notlike "Java*"    # True

# Match (regex)
"PowerShell" -match "^Power"    # True
"teste123" -match "\d+"         # True
```

 # Contains (coleções)

1,2,3,4,5 -contains 3 *# True*

"João","Maria" -notcontains "Pedro" *# True*

 # In (elemento em coleção)

3 -in 1,2,3,4,5 *# True*

"Pedro" -notin "João","Maria" *# True*

 # Operadores case-sensitive (prefixo 'c')

"PowerShell" -ceq "PowerShell" *# True (case-sensitive equal)*

"PowerShell" -cne "powershell" *# True (case-sensitive not equal)*

"PowerShell" -clike "POWER*" *# False*

"PowerShell" -cmatch "^power" *# False*

Operadores Lógicos

 # AND

($a -gt 5) -and ($b -lt 10)

 # OR

($a -eq 10) -or ($b -eq 3)

 # NOT

-not ($a -eq 10)

!($a -eq 10) *# Alternativa*

 # XOR (exclusivo)

($a -eq 10) -xor ($b -eq 3)

 # Exemplos práticos

```
if (($idade -ge 18) -and ($habilitado -eq $true)) {
   Write-Host "Acesso permitido"
}

if (($status -eq "Ativo") -or ($tipo -eq "VIP")) {
   Write-Host "Processamento prioritário"
}
```

Operadores de Atribuição

```
# Atribuição simples
$x = 10

# Atribuições compostas
$x += 5    # $x = $x + 5
$x -= 3    # $x = $x - 3
$x *= 2    # $x = $x * 2
$x /= 4    # $x = $x / 4
$x %= 3    # $x = $x % 3

# Operador de atribuição com null coalescing (PS 7+)
$nome ??= "Padrão"    # Atribui apenas se $nome for $null

# Exemplos
$total = 0
$total += 100
$total += 50
Write-Host "Total: $total"  # 150
```

Operadores Especiais

```
# Range (intervalo)
```

```
1..10          # Array de 1 a 10
10..1          # Array de 10 a 1
$inicio..$fim

# Call (invocar)
& "C:\Scripts\teste.ps1"
& { Write-Host "Script block" }

# Dot source (executar no escopo atual)
. "C:\Scripts\funcoes.ps1"

# Array subexpression
@(Get-Process)    # Garante que retorna array

# Subexpression
"O resultado é: $(2 + 2)"  # "O resultado é: 4"

# Type cast
[int]"42"        # Converte string para int
[string]123      # Converte int para string

# Comma (criar array)
$array = 1, 2, 3, 4

# Index (acessar elementos)
$array[0]        # Primeiro elemento
$array[-1]       # Último elemento
$array[1..3]     # Elementos 1, 2 e 3
```

```
# Join (juntar strings)
$nomes -join ", "  # "João, Maria, Pedro"

# Split (dividir strings)
"João,Maria,Pedro" -split ","

# Replace (substituir)
"PowerShell" -replace "Power", "Super"  # "SuperShell"

# Operador ternário (PS 7+)
$resultado = $idade -ge 18 ? "Adulto" : "Menor"

# Null coalescing (PS 7+)
$valor = $null
$padrao = $valor ?? "Padrão"  # "Padrão"

# Pipeline chain (PS 7+)
Get-Process && Write-Host "Sucesso"  # Executa se anterior tiver sucesso
Get-Process || Write-Host "Falha"    # Executa se anterior falhar
```

2.2.3 Estruturas Condicionais

If, ElseIf, Else

Sintaxe básica:

```
if (condição) {
   # código se condição verdadeira
}

if (condição) {
```

```
    # código se condição verdadeira
} else {
    # código se condição falsa
}

if (condição1) {
    # código se condição1 verdadeira
} elseif (condição2) {
    # código se condição2 verdadeira
} else {
    # código se todas falsas
}
```

Exemplos práticos:

```
# Exemplo 1: Verificação simples
$idade = 25

if ($idade -ge 18) {
    Write-Host "Você é maior de idade"
}

# Exemplo 2: If-Else
$nota = 7.5

if ($nota -ge 7) {
    Write-Host "Aprovado" -ForegroundColor Green
} else {
    Write-Host "Reprovado" -ForegroundColor Red
}
```

```
# Exemplo 3: If-ElseIf-Else (classificação)
$nota = 8.5

if ($nota -ge 9) {
   $conceito = "A"
} elseif ($nota -ge 7) {
   $conceito = "B"
} elseif ($nota -ge 5) {
   $conceito = "C"
} else {
   $conceito = "D"
}
Write-Host "Conceito: $conceito"

# Exemplo 4: Condições compostas
$saldo = 1500
$limite = 2000
$ativo = $true

if (($saldo -gt 0) -and $ativo) {
   Write-Host "Conta ativa com saldo positivo"
} elseif (($saldo -le 0) -and $ativo) {
   Write-Host "Conta ativa mas sem saldo"
} else {
   Write-Host "Conta inativa"
}
```

```
# Exemplo 5: Verificando existência
$arquivo = "C:\temp\teste.txt"

if (Test-Path $arquivo) {
   $conteudo = Get-Content $arquivo
   Write-Host "Arquivo existe. Linhas: $($conteudo.Count)"
} else {
   Write-Host "Arquivo não encontrado"
   New-Item -Path $arquivo -ItemType File
}

# Exemplo 6: Verificando tipo de objeto
$objeto = Get-Process | Select-Object -First 1

if ($objeto -is [System.Diagnostics.Process]) {
   Write-Host "É um objeto Process"
}

# Exemplo 7: Verificando nulo
$variavel = $null

if ($null -eq $variavel) {
   Write-Host "Variável é nula"
}

# Ou forma mais moderna (PS 7+)
if (-not $variavel) {
   Write-Host "Variável é nula, vazia ou false"
```

}

Switch

O Switch é ideal para múltiplas comparações com o mesmo valor.

Sintaxe básica:

```
switch (expressão) {
   valor1 { código1 }
   valor2 { código2 }
   default { código padrão }
}
```

Exemplos práticos:

```
# Exemplo 1: Switch básico
$dia = "Segunda"

switch ($dia) {
   "Segunda"  { Write-Host "Início da semana" }
   "Sexta"    { Write-Host "Quase fim de semana!" }
   "Sábado"   { Write-Host "Final de semana!" }
   "Domingo"  { Write-Host "Final de semana!" }
   default    { Write-Host "Meio da semana" }
}

# Exemplo 2: Switch com múltiplos valores
$numero = 2

switch ($numero) {
   {$_ -lt 0}    { Write-Host "Negativo" }
   0             { Write-Host "Zero" }
   {$_ -gt 0}    { Write-Host "Positivo" }
```

```
}

# Exemplo 3: Switch com arrays
$cores = "Vermelho", "Azul", "Verde"

switch ($cores) {
   "Vermelho" { Write-Host "Cor quente" }
   "Azul"     { Write-Host "Cor fria" }
   "Verde"    { Write-Host "Cor neutra" }
}

# Exemplo 4: Switch com Regex
$texto = "PowerShell123"

switch -Regex ($texto) {
   '^\w+$'    { Write-Host "Apenas letras e números" }
   '\d+'      { Write-Host "Contém dígitos" }
   '^Power'   { Write-Host "Começa com Power" }
}

# Exemplo 5: Switch com Wildcard
$arquivo = "relatorio_2025.xlsx"

switch -Wildcard ($arquivo) {
   "*.txt"   { Write-Host "Arquivo de texto" }
   "*.xlsx"  { Write-Host "Planilha Excel" }
   "*.pdf"   { Write-Host "Documento PDF" }
   default   { Write-Host "Tipo desconhecido" }
```

```
}

# Exemplo 6: Switch com File (lê linhas de arquivo)
$configFile = "C:\temp\config.txt"
"Server01","Server02","Server03" | Out-File $configFile

switch -File $configFile {
   "Server01" { Write-Host "Processando Server01" }
   "Server02" { Write-Host "Processando Server02" }
   default    { Write-Host "Servidor: $_" }
}

# Exemplo 7: Switch case-sensitive
$texto = "PowerShell"

switch -CaseSensitive ($texto) {
   "powershell" { Write-Host "Minúsculo" }
   "POWERSHELL" { Write-Host "Maiúsculo" }
   "PowerShell" { Write-Host "Capitalizado" }
}

# Exemplo 8: Switch com break
$numero = 5

switch ($numero) {
   {$_ -gt 0} {
      Write-Host "Positivo"
      # Continue para próxima avaliação
```

```
    }
    {$_ -eq 5} {
        Write-Host "É cinco"
        break  # Para execução aqui
    }
    {$_ -lt 10} {
        Write-Host "Menor que 10"  # Não será executado por causa do break
    }
}
```

Operador Ternário (PowerShell 7+)

```
# Sintaxe: condição ? valor_se_true : valor_se_false

# Exemplo 1: Atribuição simples
$idade = 20
$categoria = $idade -ge 18 ? "Adulto" : "Menor"
Write-Host $categoria  # Adulto

# Exemplo 2: Em expressões
$nota = 8
Write-Host ($nota -ge 7 ? "Aprovado" : "Reprovado")

# Exemplo 3: Aninhado
$pontos = 85
$nivel = $pontos -ge 90 ? "Ouro" : $pontos -ge 70 ? "Prata" : "Bronze"
Write-Host $nivel  # Prata

# Exemplo 4: Com operadores
$saldo = 100
```

```
$taxa = $saldo -gt 0 ? 0.05 : 0
$total = $saldo + ($saldo * $taxa)
Write-Host "Total: $total"
```

2.2.4 Estruturas de Repetição

For Loop

O loop for é usado quando você sabe quantas vezes quer iterar.

Sintaxe:

```
for (inicialização; condição; incremento) {
  # código
}
```

Exemplos práticos:

```
# Exemplo 1: Loop básico
for ($i = 0; $i -lt 10; $i++) {
  Write-Host "Iteração: $i"
}

# Exemplo 2: Loop decrescente
for ($i = 10; $i -ge 0; $i--) {
  Write-Host "Contagem regressiva: $i"
}

# Exemplo 3: Pular de 2 em 2
for ($i = 0; $i -le 20; $i += 2) {
  Write-Host "Número par: $i"
}

# Exemplo 4: Percorrer array com índice
$frutas = "Maçã", "Banana", "Laranja", "Uva"
```

```
for ($i = 0; $i -lt $frutas.Count; $i++) {
  Write-Host "[$i] $($frutas[$i])"
}

# Exemplo 5: Loop aninhado (tabuada)
for ($i = 1; $i -le 5; $i++) {
  Write-Host "` nTabuada do $i :"
  for ($j = 1; $j -le 10; $j++) {
    $resultado = $i * $j
    Write-Host "$i x $j = $resultado"
  }
}

# Exemplo 6: Múltiplas variáveis
for ($i = 0, $j = 10; $i -lt 5; $i++, $j--) {
  Write-Host "i = $i, j = $j"
}

# Exemplo 7: Processando arquivos
$arquivos = Get-ChildItem -Path C:\Temp -Filter *.txt

for ($i = 0; $i -lt $arquivos.Count; $i++) {
  Write-Host "Processando arquivo $($i+1) de $($arquivos.Count):
$($arquivos[$i].Name)"
}
```

ForEach Loop

O loop foreach itera sobre coleções de objetos.

Sintaxe:

```
foreach ($item in $colecao) {
   # código
}
```

Exemplos práticos:

```
# Exemplo 1: Iterar sobre array
$numeros = 1, 2, 3, 4, 5

foreach ($num in $numeros) {
   $quadrado = $num * $num
   Write-Host "$num ao quadrado = $quadrado"
}

# Exemplo 2: Iterar sobre processos
$processos = Get-Process | Select-Object -First 5

foreach ($proc in $processos) {
   Write-Host "Processo: $($proc.Name) - ID: $($proc.Id)"
}

# Exemplo 3: Iterar sobre arquivos
$arquivos = Get-ChildItem -Path C:\Temp -File

foreach ($arquivo in $arquivos) {
   $tamanhoKB = [math]::Round($arquivo.Length / 1KB, 2)
   Write-Host "$($arquivo.Name): $tamanhoKB KB"
}
```

```
# Exemplo 4: Iterar sobre hashtable
$configuracoes = @{
  Servidor = "192.168.1.100"
  Porta = 8080
  SSL = $true
}

foreach ($config in $configuracoes.GetEnumerator()) {
  Write-Host "$($config.Key) = $($config.Value)"
}

# Exemplo 5: Modificar elementos (nota: arrays são imutáveis)
$nomes = "joão", "maria", "pedro"
$nomesCapitalizados = @()

foreach ($nome in $nomes) {
  $nomesCapitalizados += (Get-Culture).TextInfo.ToTitleCase($nome)
}
$nomesCapitalizados

# Exemplo 6: ForEach com objetos customizados
$funcionarios = @(
  [PSCustomObject]@{Nome="João"; Salario=3000}
  [PSCustomObject]@{Nome="Maria"; Salario=3500}
  [PSCustomObject]@{Nome="Pedro"; Salario=2800}
)

foreach ($func in $funcionarios) {
```

```
    $aumento = $func.Salario * 0.10
    $novoSalario = $func.Salario + $aumento
    Write-Host "$($func.Nome): R$ $($func.Salario) → R$ $novoSalario"
}

# Exemplo 7: ForEach aninhado
$servidores = "Server01", "Server02"
$servicos = "wuauserv", "spooler"

foreach ($servidor in $servidores) {
    Write-Host "` nVerificando $servidor :"
    foreach ($servico in $servicos) {
        # Simulação - em produção usaria Invoke-Command
        Write-Host "  Verificando serviço $servico"
    }
}
```

While Loop

O loop while executa enquanto uma condição for verdadeira.

Sintaxe:

```
while (condição) {
    # código
}
```

Exemplos práticos:

```
# Exemplo 1: Contador simples
$contador = 0

while ($contador -lt 5) {
    Write-Host "Contador: $contador"
```

```
    $contador++
}

# Exemplo 2: Aguardar condição
$processo = Get-Process notepad -ErrorAction SilentlyContinue

while ($processo) {
    Write-Host "Notepad ainda está rodando..."
    Start-Sleep -Seconds 2
    $processo = Get-Process notepad -ErrorAction SilentlyContinue
}
Write-Host "Notepad foi fechado"

# Exemplo 3: Processar até entrada específica
$continuar = $true

while ($continuar) {
    $resposta = Read-Host "Digite 'sair' para terminar"
    if ($resposta -eq "sair") {
        $continuar = $false
    } else {
        Write-Host "Você digitou: $resposta"
    }
}

# Exemplo 4: Tentativas com limite
$tentativas = 0
$maxTentativas = 3
```

```
$sucesso = $false

while (($tentativas -lt $maxTentativas) -and (-not $sucesso)) {
    $tentativas++
    Write-Host "Tentativa $tentativas de $maxTentativas"

    # Simulação de operação
    $resultado = Get-Random -Minimum 1 -Maximum 10
    if ($resultado -gt 5) {
        $sucesso = $true
        Write-Host "Sucesso!"
    } else {
        Write-Host "Falha. Tentando novamente..."
    }
}

if (-not $sucesso) {
    Write-Host "Todas as tentativas falharam"
}

# Exemplo 5: Processar fila
$fila = 1..10

while ($fila.Count -gt 0) {
    $item = $fila[0]
    Write-Host "Processando item: $item"
    $fila = $fila[1..($fila.Count-1)]
    Start-Sleep -Milliseconds 500
```

```
}

# Exemplo 6: Monitoramento
$limite = 80
$uso = (Get-Counter '\Processor(_Total)\% Processor
Time').CounterSamples.CookedValue

while ($uso -lt $limite) {
  Write-Host "Uso de CPU: $([math]::Round($uso,2))% (Limite: $limite%)"
  Start-Sleep -Seconds 5
  $uso = (Get-Counter '\Processor(_Total)\% Processor
Time').CounterSamples.CookedValue
}
Write-Host "ALERTA: CPU acima do limite!"
```

Do-While e Do-Until

Executam o bloco **pelo menos uma vez** antes de verificar a condição.

Sintaxe:

```
# Do-While: repete enquanto condição for verdadeira
do {
  # código
} while (condição)

# Do-Until: repete até condição ser verdadeira
do {
  # código
} until (condição)
```

Exemplos práticos:

```
# Exemplo 1: Do-While básico
$numero = 0
```

```
do {
   Write-Host "Número: $numero"
   $numero++
} while ($numero -lt 5)

# Exemplo 2: Do-Until básico
$contador = 0

do {
   Write-Host "Tentativa: $contador"
   $contador++
} until ($contador -eq 5)

# Exemplo 3: Menu interativo com Do-While
do {
   Write-Host "`n===== MENU ====="
   Write-Host "1. Opção 1"
   Write-Host "2. Opção 2"
   Write-Host "3. Sair"
   $opcao = Read-Host "Escolha uma opção"

   switch ($opcao) {
      "1" { Write-Host "Opção 1 selecionada" }
      "2" { Write-Host "Opção 2 selecionada" }
      "3" { Write-Host "Saindo..." }
      default { Write-Host "Opção inválida" }
   }
```

```
} while ($opcao -ne "3")

# Exemplo 4: Validação de entrada
do {
    $idade = Read-Host "Digite sua idade (18-100)"
    $idadeNum = [int]$idade
} while (($idadeNum -lt 18) -or ($idadeNum -gt 100))

Write-Host "Idade válida: $idadeNum"

# Exemplo 5: Repetir até sucesso
$tentativa = 0
do {
    $tentativa++
    Write-Host "Tentando conectar... (Tentativa $tentativa)"
    $conexao = Test-Connection "8.8.8.8" -Count 1 -Quiet
    if (-not $conexao) {
        Start-Sleep -Seconds 2
    }
} until ($conexao)

Write-Host "Conexão estabelecida!"

# Exemplo 6: Processar até lista vazia
$tarefas = @("Tarefa 1", "Tarefa 2", "Tarefa 3")

do {
    $tarefaAtual = $tarefas[0]
```

```
    Write-Host "Executando: $tarefaAtual"
    $tarefas = $tarefas[1..($tarefas.Count-1)]
} while ($tarefas.Count -gt 0)

Write-Host "Todas as tarefas concluídas!"
```

Break e Continue

Controlam o fluxo dentro dos loops.

Break: Sai completamente do loop

```
# Exemplo 1: Break simples
for ($i = 1; $i -le 10; $i++) {
    if ($i -eq 5) {
        break
    }
    Write-Host $i
}
# Saída: 1, 2, 3, 4

# Exemplo 2: Break em While
$contador = 0
while ($true) {
    $contador++
    Write-Host "Iteração: $contador"

    if ($contador -eq 3) {
        Write-Host "Atingiu o limite, saindo..."
        break
    }
}
```

```
# Exemplo 3: Break com label (loops aninhados)
:exterior for ($i = 1; $i -le 3; $i++) {
  for ($j = 1; $j -le 3; $j++) {
    Write-Host "i=$i, j=$j"
    if ($i -eq 2 -and $j -eq 2) {
      break exterior  # Sai de ambos os loops
    }
  }
}

# Exemplo 4: Buscar e parar quando encontrar
$numeros = 1..100
$alvo = 42

foreach ($num in $numeros) {
  Write-Host "Verificando: $num"
  if ($num -eq $alvo) {
    Write-Host "Encontrado: $alvo"
    break
  }
}
```

Continue: Pula para próxima iteração

```
# Exemplo 1: Continue simples
for ($i = 1; $i -le 10; $i++) {
  if ($i % 2 -eq 0) {
    continue  # Pula números pares
  }
```

```
    Write-Host $i
}
# Saída: 1, 3, 5, 7, 9

# Exemplo 2: Continue em ForEach
$arquivos = Get-ChildItem -Path C:\Temp

foreach ($arquivo in $arquivos) {
    if ($arquivo.Extension -ne ".txt") {
        continue  # Pula se não for .txt
    }
    Write-Host "Processando: $($arquivo.Name)"
}

# Exemplo 3: Continue com validação
$numeros = 1, "texto", 3, $null, 5, "erro", 7

foreach ($item in $numeros) {
    if ($item -isnot [int]) {
        Write-Host "Pulando item inválido: $item"
        continue
    }
    $resultado = $item * 2
    Write-Host "$item x 2 = $resultado"
}

# Exemplo 4: Continue com múltiplas condições
$funcionarios = @(
```

```
    @{Nome="João"; Idade=25; Ativo=$true}
    @{Nome="Maria"; Idade=17; Ativo=$true}
    @{Nome="Pedro"; Idade=30; Ativo=$false}
    @{Nome="Ana"; Idade=28; Ativo=$true}
)

foreach ($func in $funcionarios) {
    if ($func.Idade -lt 18) {
        Write-Host "Pulando $($func.Nome) - Menor de idade"
        continue
    }
    if (-not $func.Ativo) {
        Write-Host "Pulando $($func.Nome) - Inativo"
        continue
    }
    Write-Host "Processando funcionário: $($func.Nome)"
}
```

Comparação entre Loops

Loop	Quando usar	Vantagens
for	Número conhecido de iterações	Controle preciso do contador
foreach	Percorrer coleções	Sintaxe simples, mais legível
while	Condição pode ser falsa desde início	Pode não executar nenhuma vez
do-while	Deve executar pelo menos uma vez	Garantia de execução mínima
do-until	Lógica inversa do while	Mais intuitivo em alguns casos

Conclusão da Seção 2

Nesta seção, exploramos em profundidade os fundamentos da sintaxe PowerShell:

1. **Cmdlets**: Estrutura Verbo-Substantivo, parâmetros e descoberta de comandos
2. **Pipeline**: O coração do PowerShell, passando objetos entre comandos
3. **Variáveis e Operadores**: Tipos de dados, escopos e operações
4. **Estruturas Condicionais**: If, Switch e operador ternário
5. **Estruturas de Repetição**: For, ForEach, While, Do-While/Until
6. **Controle de Fluxo**: Break e Continue

Com esses conceitos dominados, você tem a base necessária para escrever

scripts PowerShell eficientes e resolver problemas complexos de automação.

3. MANIPULAÇÃO DE OBJETOS E DADOS

3.1 Objetos, Propriedades e Métodos

3.1.1 Paradigma Orientado a Objetos no PowerShell

O que são Objetos?

No PowerShell, **tudo é um objeto**. Esta é a diferença fundamental entre PowerShell e shells tradicionais como Bash ou CMD, que trabalham com texto puro.

Conceito:

Um objeto é uma estrutura de dados que combina:

- **Propriedades**: Características ou atributos do objeto (dados)
- **Métodos**: Ações que o objeto pode executar (comportamentos)
- **Tipo**: Classificação do objeto na hierarquia .NET

Analogia do mundo real:

Objeto: Carro

├── Propriedades (características)

│ ├── Marca: "Toyota"

│ ├── Modelo: "Corolla"

│ ├── Ano: 2024

│ ├── Cor: "Prata"

│ └── Velocidade: 0

└── Métodos (ações)

├── Ligar()

├── Desligar()

├── Acelerar()

└── Frear()

Exemplo prático no PowerShell:

```
# Obter um processo (objeto)
$processo = Get-Process -Name powershell | Select-Object -First 1

# O processo é um objeto com propriedades e métodos
Write-Host "Tipo do objeto: $($processo.GetType().FullName)"
# System.Diagnostics.Process

# Acessar propriedades (dados)
Write-Host "Nome: $($processo.ProcessName)"
Write-Host "ID: $($processo.Id)"
Write-Host "Memória: $($processo.WorkingSet / 1MB) MB"

# Chamar métodos (ações)
# $processo.Kill()  # Encerra o processo
# $processo.Refresh()  # Atualiza os dados
```

Diferença entre Texto e Objetos

Shells tradicionais (texto):

 # Linux/Bash - retorna TEXTO

$ ps aux | grep firefox

user   12345 2.5 3.2 /usr/bin/firefox

 # Para extrair o PID, é necessário parsing de texto

$ ps aux | grep firefox | awk '{print $2}'

PowerShell (objetos):

 # PowerShell - retorna OBJETOS

Get-Process -Name firefox

 # Acesso direto à propriedade, sem parsing

$firefox = Get-Process -Name firefox

$firefox.Id *# Acesso direto ao PID como número*

 # Operações matemáticas diretas

$firefox.WorkingSet / 1MB *# Memória em MB*

Vantagens dos objetos:

- ✅ Acesso direto a propriedades
- ✅ Tipos de dados preservados
- ✅ Sem necessidade de parsing de texto
- ✅ IntelliSense e autocompletar
- ✅ Validação de tipos em tempo de execução
- ✅ Métodos disponíveis para manipulação

3.1.2 Explorando Objetos com Get-Member

O cmdlet **Get-Member** é essencial para descobrir propriedades e métodos de objetos.

Sintaxe e Uso Básico

 # Sintaxe

Get-Command | Get-Member

objeto | Get-Member [parâmetros]

 # Ver todos os membros de um objeto

Get-Process | Get-Member

 # Filtrar por tipo de membro

Get-Process | Get-Member -MemberType Property

Get-Process | Get-Member -MemberType Method

Get-Process | Get-Member -MemberType Event

 # Buscar membro específico

Get-Process | Get-Member -Name *Memory*

 # Ver membros estáticos

Get-Process | Get-Member -Static

Tipos de Membros

1. Properties (Propriedades)

Armazenam dados/características do objeto:

$processo = Get-Process | Select-Object -First 1

 # Ver todas as propriedades

$processo | Get-Member -MemberType Property

 # Propriedades comuns de Process:

$processo.Id *# Int32 - ID do processo*

$processo.ProcessName *# String - Nome do processo*

$processo.StartTime *# DateTime - Hora de início*

$processo.CPU *# Double - Tempo de CPU*

$processo.WorkingSet *# Int64 - Memória física*

$processo.Threads *# ProcessThreadCollection - Threads*

$processo.Handles *# Int32 - Número de handles*

Tipos de propriedades:

 # Properties (leitura e escrita)

$processo.PriorityClass = 'High'

 # Nota: Muitas propriedades são read-only

 # $processo.Id = 999 # ❌ Erro - somente leitura

2. Methods (Métodos)

Executam ações ou retornam valores calculados:

$processo = Get-Process -Name notepad | Select-Object -First 1

 # Ver todos os métodos

$processo | Get-Member -MemberType Method

 # Métodos comuns de Process:

$processo.Kill() *# Encerra o processo*

$processo.CloseMainWindow() *# Fecha janela principal*

$processo.Refresh() *# Atualiza dados do objeto*

$processo.WaitForExit() *# Aguarda encerramento*

$processo.ToString() *# Converte para string*

 # Métodos podem ter parâmetros

$processo.WaitForExit(5000) *# Aguarda até 5 segundos*

Diferença entre propriedade e método:

 # Propriedade - acesso direto, sem parênteses

$processo.ProcessName

 # Método - execução de código, COM parênteses

$processo.ToString()

$processo.GetType()

 # Erro comum: esquecer parênteses em método

$processo.ToString *# ❌ Retorna informações do método, não executa*

$processo.ToString() *# ✅ Executa o método*

3. Script Properties

Propriedades calculadas dinamicamente:

$arquivo = Get-ChildItem | Select-Object -First 1

$arquivo | Get-Member -MemberType ScriptProperty

 # Exemplos de ScriptProperties:

$arquivo.PSChildName *# Nome do item*

$arquivo.PSDrive *# Drive onde está localizado*

$arquivo.PSIsContainer *# É um diretório?*

$arquivo.PSPath *# Caminho completo do provider*

4. Nota Properties

Propriedades adicionadas dinamicamente por cmdlets:

 # Algumas propriedades são adicionadas por formatação

Get-Process | Format-Table | Get-Member -MemberType NoteProperty

5. Alias Properties

Nomes alternativos para propriedades existentes:

$arquivo = Get-ChildItem | Select-Object -First 1

$arquivo | Get-Member -MemberType AliasProperty

 # Exemplo: 'Name' pode ser um alias para 'PSChildName'

6. Events (Eventos)

Notificações que objetos podem disparar:

$processo = Get-Process | Select-Object -First 1

$processo | Get-Member -MemberType Event

 # Eventos comuns de Process:

 # - Exited: Disparado quando processo termina

 # - Disposed: Disparado quando objeto é liberado

Anatomia Completa de um Objeto

 # Criar um objeto para análise

$arquivo = Get-ChildItem C:\Windows\notepad.exe

 # 1. Ver o tipo do objeto

$arquivo.GetType()

 # Retorna: System.IO.FileInfo

 # 2. Ver hierarquia de herança

$arquivo.GetType().BaseType

 # FileSystemInfo -> MarshalByRefObject -> Object

 # 3. Ver todos os membros

$arquivo | Get-Member

 # 4. Propriedades mais comuns

$arquivo.Name *# Nome do arquivo*

$arquivo.FullName *# Caminho completo*

$arquivo.Length *# Tamanho em bytes*

```
$arquivo.Extension     # Extensão (.exe)
$arquivo.Directory     # Objeto DirectoryInfo
$arquivo.CreationTime  # Data de criação
$arquivo.LastWriteTime # Última modificação
$arquivo.Attributes    # Atributos (Hidden, ReadOnly, etc)

# 5. Métodos mais comuns
$arquivo.Delete()        # Exclui o arquivo
$arquivo.MoveTo($destino)  # Move o arquivo
$arquivo.CopyTo($destino)  # Copia o arquivo
$arquivo.Refresh()       # Atualiza informações
$arquivo.ToString()      # Converte para string
$arquivo.GetHashCode()   # Retorna hash code

# 6. Métodos de extensão do PowerShell
$arquivo | Get-Content   # Lê conteúdo (arquivo texto)
$arquivo | Remove-Item   # Remove o arquivo
```

3.1.3 Acessando Propriedades e Métodos

Notação de Ponto (Dot Notation)

```
# Sintaxe: objeto.propriedade ou objeto.método()

# Acessar propriedade
$processo = Get-Process | Select-Object -First 1
$nome = $processo.ProcessName
$memoria = $processo.WorkingSet

# Chamar método
$tipo = $processo.GetType()
```

```
$texto = $processo.ToString()

# Encadear (chaining)
$processo.StartTime.ToString("dd/MM/yyyy HH:mm:ss")
$arquivo.Directory.FullName
$texto.ToUpper().Substring(0, 5)

# Acessar propriedade de propriedade
$processo.Threads.Count
$processo.MainModule.FileName
```

Propriedades Aninhadas

```
# Objetos podem conter outros objetos
$servico = Get-Service | Select-Object -First 1

# Propriedade simples
$servico.Name

# Propriedade que é um objeto
$servico.ServicesDependedOn    # Array de objetos ServiceController

# Acessar propriedade do objeto aninhado
$servico.ServicesDependedOn[0].Name

# Percorrer coleção aninhada
$servico.ServicesDependedOn | ForEach-Object {
  Write-Host "Dependência: $($_.Name)"
}
```

Operador de Acesso de Membro (Member Access Operator)

```
# Quando o nome da propriedade está em uma variável
$propriedade = "ProcessName"
$processo = Get-Process | Select-Object -First 1

# Não funciona:
# $processo.$propriedade  # ❌ Tenta acessar literal "$propriedade"

# Funciona:
$processo.$propriedade  # ✅ Em alguns casos funciona
$processo.($propriedade)  # ✅ Sintaxe explícita, sempre funciona

# Exemplo prático
$propriedades = "Name", "Id", "CPU"
$processo = Get-Process | Select-Object -First 1

foreach ($prop in $propriedades) {
   $valor = $processo.($prop)
   Write-Host "$prop : $valor"
}
```

Métodos com Parâmetros

```
# Métodos podem aceitar parâmetros

# Método sem parâmetros
$texto = "PowerShell"
$texto.ToUpper()      # "POWERSHELL"

# Método com um parâmetro
$texto.Substring(5)    # "Shell"
```

 # Método com múltiplos parâmetros

$texto.Substring(0, 5) *# "Power"*

 # Método com parâmetros nomeados (.NET style)

$arquivo.CopyTo("C:\destino\arquivo.txt", $true) *# $true = sobrescrever*

 # Descobrir assinatura de método

$texto | Get-Member -Name Substring

 # Mostrará todas as sobrecargas (overloads)

Métodos Estáticos

Métodos que pertencem à classe, não à instância:

 # Sintaxe: [Tipo]::Método()

 # Exemplos comuns

[Math]::Round(3.14159, 2) *# 3.14*

[Math]::Sqrt(16) *# 4*

[Math]::Max(10, 20) *# 20*

[Math]::Min(10, 20) *# 10*

[Math]::Abs(-42) *# 42*

[Math]::Pow(2, 8) *# 256*

[String]::IsNullOrEmpty($texto) *# True/False*

[String]::Join(", ", $array) *# Une array em string*

[DateTime]::Now *# Data/hora atual*

[DateTime]::Today *# Data atual (00:00:00)*

[DateTime]::ParseExact($string, $formato, $cultura)

```
[System.IO.Path]::GetFileName($caminho)
[System.IO.Path]::GetExtension($caminho)
[System.IO.Path]::Combine($parte1, $parte2)

[System.Environment]::MachineName   # Nome do computador
[System.Environment]::UserName      # Nome do usuário
[System.Environment]::OSVersion     # Versão do SO

# Listar métodos estáticos
[Math] | Get-Member -Static
[DateTime] | Get-Member -Static
```

3.1.4 Criando Objetos Customizados

PSCustomObject (Recomendado)

A forma moderna e recomendada de criar objetos:

```
# Sintaxe básica
$objeto = [PSCustomObject]@{
   Propriedade1 = Valor1
   Propriedade2 = Valor2
}

# Exemplo 1: Objeto simples
$pessoa = [PSCustomObject]@{
   Nome = "João Silva"
   Idade = 30
   Cidade = "São Paulo"
   Ativo = $true
}
```

```
# Acessar propriedades
$pessoa.Nome       # "João Silva"
$pessoa.Idade      # 30

# Modificar propriedades
$pessoa.Idade = 31

# Exibir
$pessoa | Format-Table
$pessoa | Format-List

# Exemplo 2: Múltiplos objetos
$funcionarios = @(
  [PSCustomObject]@{
    Nome = "Maria"
    Cargo = "Gerente"
    Salario = 8000
  },
  [PSCustomObject]@{
    Nome = "Pedro"
    Cargo = "Analista"
    Salario = 5000
  },
  [PSCustomObject]@{
    Nome = "Ana"
    Cargo = "Desenvolvedor"
    Salario = 6000
```

```
    }
)

# Manipular coleção
$funcionarios | Where-Object Salario -gt 5500
$funcionarios | Sort-Object Salario -Descending
$funcionarios | Select-Object Nome, Cargo

# Exemplo 3: Objeto com propriedades calculadas
$servidor = [PSCustomObject]@{
    Nome = $env:COMPUTERNAME
    SO = [System.Environment]::OSVersion.VersionString
    CPU = (Get-CimInstance Win32_Processor).Name
    MemoriaGB = [math]::Round((Get-CimInstance
Win32_ComputerSystem).TotalPhysicalMemory / 1GB, 2)
    DataConsulta = Get-Date
}

$servidor | Format-List
```

New-Object (Legado)

Forma antiga, menos eficiente:

```
# Criar objeto COM
$shell = New-Object -ComObject WScript.Shell
$shell.Popup("Mensagem de teste")

# Criar objeto .NET
$lista = New-Object System.Collections.ArrayList
$lista.Add("Item 1")
```

```
$lista.Add("Item 2")


# Criar PSObject (forma antiga)
$objeto = New-Object PSObject -Property @{
  Nome = "Teste"
  Valor = 100
}


# ⚠️ Nota: Prefira [PSCustomObject] em código novo
```

Adicionando Propriedades a Objetos Existentes

```
# Add-Member: adiciona propriedades/métodos dinamicamente


# Exemplo 1: Adicionar propriedade simples
$processo = Get-Process | Select-Object -First 1
$processo | Add-Member -MemberType NoteProperty -Name "Categoria" -Value "Sistema"
$processo.Categoria  # "Sistema"


# Exemplo 2: Adicionar propriedade calculada
$arquivo = Get-ChildItem | Select-Object -First 1
$arquivo | Add-Member -MemberType ScriptProperty -Name "TamanhoMB" -Value {
  [math]::Round($this.Length / 1MB, 2)
}
$arquivo.TamanhoMB


# Exemplo 3: Adicionar método
$objeto = [PSCustomObject]@{Nome = "Teste"}
$objeto | Add-Member -MemberType ScriptMethod -Name "Saudar" -Value {
```

```
    "Olá, $($this.Nome)!"
}
$objeto.Saudar()  # "Olá, Teste!"

# Exemplo 4: Adicionar alias
$pessoa = [PSCustomObject]@{
    PrimeiroNome = "João"
    Sobrenome = "Silva"
}
$pessoa | Add-Member -MemberType AliasProperty -Name "Nome" -Value "PrimeiroNome"
$pessoa.Nome  # "João"

# Exemplo 5: Enriquecer objetos em pipeline
Get-Process | Select-Object -First 3 | ForEach-Object {
    $_ | Add-Member -MemberType NoteProperty -Name "MemoriaMB" -Value ([math]::Round($_.WorkingSet / 1MB, 2))
    $_
} | Format-Table Name, Id, MemoriaMB
```

Select-Object para Criar Objetos

```
# Select-Object pode criar objetos com propriedades específicas

# Exemplo 1: Selecionar propriedades existentes
Get-Process | Select-Object Name, Id, CPU

# Exemplo 2: Propriedades calculadas
Get-Process | Select-Object Name,
    @{Name='CPUTime'; Expression={$_.CPU}},
    @{Name='MemoryMB'; Expression={[math]::Round($_.WorkingSet / 1MB, 2)}}
```

```
# Exemplo 3: Múltiplas propriedades calculadas
Get-ChildItem | Select-Object Name,
   @{N='SizeMB'; E={'{0:N2}' -f ($_.Length / 1MB)}},
   @{N='Modified'; E={$_.LastWriteTime.ToString('dd/MM/yyyy')}},
   @{N='IsLarge'; E={$_.Length -gt 1MB}}

# Exemplo 4: Criar objeto completamente novo
$dados = Get-Process | Select-Object @{
   Name = 'Resumo'
   Expression = {"$($_.Name) - PID: $($_.Id)"}
}

# Exemplo 5: Combinar com Group-Object
Get-Service | Group-Object Status | Select-Object @{
   Name = 'Status'
   Expression = {$_.Name}
}, @{
   Name = 'Quantidade'
   Expression = {$_.Count}
}, @{
   Name = 'Servicos'
   Expression = {$_.Group.Name -join ', '}
}
```

3.1.5 Trabalhando com Coleções

Arrays

```
# Criar arrays
$numeros = 1, 2, 3, 4, 5
```

```
$nomes = @("João", "Maria", "Pedro")
$vazio = @()
$range = 1..10

# Acessar elementos
$numeros[0]      # Primeiro elemento (1)
$numeros[-1]     # Último elemento (5)
$numeros[1..3]   # Elementos 1, 2, 3
$numeros[-3..-1]  # Últimos 3 elementos

# Propriedades
$numeros.Count    # Número de elementos
$numeros.Length   # Mesmo que Count

# Arrays são imutáveis (tamanho fixo)
$numeros += 6     # Cria um NOVO array

# Iterar
foreach ($num in $numeros) {
  Write-Host $num
}

# Métodos
$nomes.Contains("João")      # True
[Array]::IndexOf($nomes, "Maria")  # 1
[Array]::Reverse($numeros)
[Array]::Sort($nomes)
```

ArrayList (Coleção Dinâmica)

```
# Criar ArrayList (mutável)
$lista = New-Object System.Collections.ArrayList

# Adicionar elementos
$lista.Add("Item 1")
$lista.Add("Item 2")
$lista.Add("Item 3")

# Ou usando casting
$lista = [System.Collections.ArrayList]@()
[void]$lista.Add("Item 1")  # [void] suprime saída do índice

# Inserir em posição
$lista.Insert(1, "Item 1.5")

# Remover
$lista.Remove("Item 2")     # Remove por valor
$lista.RemoveAt(0)          # Remove por índice
$lista.Clear()              # Remove todos

# Propriedades e métodos
$lista.Count
$lista.Contains("Item 1")
$lista.IndexOf("Item 3")
```

Hashtables

```
# Criar hashtable
$config = @{
  Servidor = "192.168.1.100"
```

```
    Porta = 8080
    SSL = $true
    Timeout = 30
}

# Acessar valores
$config["Servidor"]
$config.Servidor      # Sintaxe alternativa

# Adicionar/modificar
$config["Usuario"] = "admin"
$config.Senha = "senha123"

# Remover
$config.Remove("Senha")

# Verificar existência
$config.ContainsKey("Servidor")   # True
$config.ContainsValue(8080)       # True

# Iterar
foreach ($item in $config.GetEnumerator()) {
    Write-Host "$($item.Key) = $($item.Value)"
}

# Propriedades e métodos
$config.Keys         # Coleção de chaves
$config.Values       # Coleção de valores
```

```
$config.Count        # Número de itens

# Hashtable ordenada
$ordenado = [ordered]@{
  Primeiro = 1
  Segundo = 2
  Terceiro = 3
}
```

Generic Lists (Coleções Tipadas)

```
# List<T> - coleção tipada e performática

# Criar lista de strings
$nomes = [System.Collections.Generic.List[string]]::new()
$nomes.Add("João")
$nomes.Add("Maria")

# Criar lista de inteiros
$numeros = [System.Collections.Generic.List[int]]::new()
$numeros.Add(10)
$numeros.Add(20)

# Criar lista de objetos customizados
$pessoas = [System.Collections.Generic.List[PSCustomObject]]::new()
$pessoas.Add([PSCustomObject]@{Nome="João"; Idade=30})
$pessoas.Add([PSCustomObject]@{Nome="Maria"; Idade=25})

# Métodos disponíveis
$nomes.Contains("João")
```

```
$nomes.IndexOf("Maria")
$nomes.Remove("João")
$nomes.Clear()
$nomes.Sort()

# Vantagens: tipagem forte, performance, IntelliSense
```

3.1.6 Conversão e Comparação de Objetos

Type Casting (Conversão de Tipos)

```
# Conversão explícita
$texto = "42"
$numero = [int]$texto       # 42 (Int32)

$numeroDecimal = "3.14"
$double = [double]$numeroDecimal  # 3.14

$dataTexto = "2025-10-15"
$data = [datetime]$dataTexto

# Conversões comuns
[string]123             # "123"
[int]"456"              # 456
[bool]1                 # True
[bool]0                 # False
[char]65                # 'A'

# Conversão com validação
try {
   $valor = [int]"abc"     # Lança exceção
```

```
}
catch {
  Write-Host "Conversão inválida"
}

# Operador -as (conversão segura)
$resultado = "abc" -as [int]  # $null se falhar (não lança exceção)
if ($resultado -eq $null) {
  Write-Host "Conversão falhou"
}
```

Verificação de Tipo

```
# Operador -is
$numero = 42
$numero -is [int]          # True
$numero -is [string]       # False

$processo = Get-Process | Select-Object -First 1
$processo -is [System.Diagnostics.Process]  # True

# Operador -isnot
$texto = "PowerShell"
$texto -isnot [int]        # True

# GetType()
$objeto = Get-Date
$objeto.GetType()          # System.DateTime
$objeto.GetType().FullName    # Nome completo do tipo
$objeto.GetType().BaseType    # Tipo base
```

 # Verificar em estruturas condicionais

```
if ($variavel -is [array]) {
   Write-Host "É um array"
}
elseif ($variavel -is [hashtable]) {
   Write-Host "É uma hashtable"
}
```

Compare-Object

 # Comparar dois conjuntos de objetos

 # Exemplo 1: Comparar arrays simples

```
$lista1 = "A", "B", "C", "D"
$lista2 = "B", "C", "D", "E"

Compare-Object -ReferenceObject $lista1 -DifferenceObject $lista2
```

 # Mostra diferenças:

 # <= (só em Reference)

 # => (só em Difference)

 # Exemplo 2: Incluir itens iguais

```
Compare-Object $lista1 $lista2 -IncludeEqual
```

 # == (em ambos)

 # Exemplo 3: Comparar processos

```
$antes = Get-Process
Start-Process notepad
$depois = Get-Process
```

Compare-Object $antes $depois -Property Name

 # Exemplo 4: Comparar arquivos

$origem = Get-ChildItem C:\Origem

$destino = Get-ChildItem C:\Destino

Compare-Object $origem $destino -Property Name, Length

 # Exemplo 5: Apenas mostrar diferenças

$dif = Compare-Object $lista1 $lista2 -PassThru

$somenteEm1 = $dif | Where-Object {$_.SideIndicator -eq '<='}

$somenteEm2 = $dif | Where-Object {$_.SideIndicator -eq '=>'}

Where-Object vs .Where()

 # Where-Object (cmdlet)

Get-Process | Where-Object CPU -gt 10

Get-Process | Where-Object {$_.WorkingSet -gt 100MB}

 # .Where() (método de array - PS 4.0+)

$processos = Get-Process

$processos.Where({$_.CPU -gt 10})

 # Diferenças:

 # - .Where() é mais rápido para grandes coleções

 # - .Where() não funciona no pipeline direto

 # - .Where() tem opções adicionais

 # Opções do .Where()

```
$numeros = 1..10

# Default: retorna todos que atendem condição
$numeros.Where({$_ -gt 5})

# First: retorna primeiro que atende
$numeros.Where({$_ -gt 5}, 'First')

# First N: retorna primeiros N que atendem
$numeros.Where({$_ -gt 5}, 'First', 3)

# Last: retorna último que atende
$numeros.Where({$_ -gt 5}, 'Last')

# SkipUntil: pula até encontrar
$numeros.Where({$_ -gt 5}, 'SkipUntil')

# Until: retorna até encontrar
$numeros.Where({$_ -gt 5}, 'Until')

# Split: separa em dois grupos
$resultado = $numeros.Where({$_ -gt 5}, 'Split')
$maiores = $resultado[0]
$menores = $resultado[1]
```

3.1.7 Exemplos Práticos Avançados

Exemplo 1: Relatório de Processos

```
# Criar relatório detalhado de processos
```

```
$relatorio = Get-Process | Where-Object {$_.WorkingSet -gt 50MB} | ForEach-Object {
  [PSCustomObject]@{
    Processo = $_.ProcessName
    PID = $_.Id
    'Memória (MB)' = [math]::Round($_.WorkingSet / 1MB, 2)
    'CPU (s)' = [math]::Round($_.CPU, 2)
    Threads = $_.Threads.Count
    'Tempo Execução' = if ($_.StartTime) {
      (Get-Date) - $_.StartTime | Select-Object -ExpandProperty TotalHours | ForEach-Object {[math]::Round($_, 2)}
    } else {
      "N/A"
    }
    Empresa = $_.Company
    Caminho = $_.Path
  }
}

# Exibir ordenado por memória
$relatorio | Sort-Object 'Memória (MB)' -Descending | Format-Table -AutoSize

# Exportar para CSV
$relatorio | Export-Csv "processos_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv" -NoTypeInformation
```

Exemplo 2: Inventário de Sistema

```
# Coletar informações detalhadas do sistema
$inventario = [PSCustomObject]@{
  # Informações do computador
```

```
    NomeComputador = $env:COMPUTERNAME
    Usuario = $env:USERNAME
    DataColeta = Get-Date

    # Sistema Operacional
    SO = (Get-CimInstance Win32_OperatingSystem).Caption
    Versao = [System.Environment]::OSVersion.Version.ToString()
    Arquitetura = (Get-CimInstance Win32_OperatingSystem).OSArchitecture

    # Hardware
    Processador = (Get-CimInstance Win32_Processor).Name
    NucleosFisicos = (Get-CimInstance Win32_Processor).NumberOfCores
    NucleosLogicos = (Get-CimInstance
Win32_Processor).NumberOfLogicalProcessors

    # Memória
    MemoriaTotalGB = [math]::Round((Get-CimInstance
Win32_ComputerSystem).TotalPhysicalMemory / 1GB, 2)
    MemoriaLivreGB = [math]::Round((Get-CimInstance
Win32_OperatingSystem).FreePhysicalMemory / 1MB, 2)

    # Disco
    Discos = (Get-CimInstance Win32_LogicalDisk -Filter "DriveType=3") | ForEach-
Object {
        [PSCustomObject]@{
            Letra = $_.DeviceID
            TamanhoGB = [math]::Round($_.Size / 1GB, 2)
            LivreGB = [math]::Round($_.FreeSpace / 1GB, 2)
            PercentualLivre = [math]::Round(($_.FreeSpace / $_.Size) * 100, 2)
```

```
    }
  }

  # Rede
  Adaptadores = (Get-NetAdapter | Where-Object Status -eq 'Up').Name -join ', '
  EnderecoIP = (Get-NetIPAddress -AddressFamily IPv4 | Where-Object {$_.InterfaceAlias -notlike '*Loopback*'}).IPAddress -join ', '
}

# Exibir
$inventario | Format-List
$inventario.Discos | Format-Table -AutoSize
```

Exemplo 3: Monitoramento de Serviços

```
# Monitorar serviços críticos
$servicosCriticos = "wuauserv", "BITS", "Spooler", "W32Time", "WinRM"

$status = foreach ($servico in $servicosCriticos) {
  $svc = Get-Service -Name $servico -ErrorAction SilentlyContinue

  [PSCustomObject]@{
    Servico = $servico
    DisplayName = if ($svc) { $svc.DisplayName } else { "Não encontrado" }
    Status = if ($svc) { $svc.Status } else { "N/A" }
    TipoInicio = if ($svc) { $svc.StartType } else { "N/A" }
    Alerta = if ($svc -and $svc.Status -ne 'Running') { " ⚠️ ATENÇÃO" } else { " ✅ OK" }
    DataVerificacao = Get-Date -Format "dd/MM/yyyy HH:mm:ss"
  }
}
```

```
# Exibir com cores
$status | ForEach-Object {
  $cor = if ($_.Alerta -like "*ATENÇÃO*") { 'Red' } else { 'Green' }
  Write-Host "$($_.Servico) - $($_.Status)" -ForegroundColor $cor
}

# Salvar log
$status | Export-Csv "status_servicos.csv" -NoTypeInformation -Append
```

3.2 Importação/Exportação de Dados

3.2.1 Trabalhando com CSV (Comma-Separated Values)

Export-Csv - Exportando para CSV

```
# Sintaxe básica
objeto | Export-Csv -Path caminho.csv

# Exemplo 1: Exportar processos
Get-Process | Export-Csv processos.csv

# Problema: inclui informação de tipo
# Solução: usar -NoTypeInformation
Get-Process | Export-Csv processos.csv -NoTypeInformation

# Exemplo 2: Especificar delimitador
Get-Process | Export-Csv processos.csv -Delimiter ';' -NoTypeInformation

# Exemplo 3: Especificar encoding
Get-Service | Export-Csv servicos.csv -Encoding UTF8 -NoTypeInformation
```

```
# Exemplo 4: Append (adicionar ao arquivo existente)
Get-Process -Name powershell | Export-Csv log.csv -Append -NoTypeInformation

# Exemplo 5: Selecionar propriedades específicas
Get-Process | Select-Object Name, Id, CPU, WorkingSet |
  Export-Csv processos_simples.csv -NoTypeInformation

# Exemplo 6: Com propriedades calculadas
Get-Process | Select-Object Name,
  @{N='MemoryMB'; E={[math]::Round($_.WorkingSet / 1MB, 2)}},
  @{N='CPUTime'; E={[math]::Round($_.CPU, 2)}} |
  Export-Csv processos_formatado.csv -NoTypeInformation

# Exemplo 7: Exportar objetos customizados
$dados = @(
  [PSCustomObject]@{Nome="João"; Idade=30; Cidade="SP"}
  [PSCustomObject]@{Nome="Maria"; Idade=25; Cidade="RJ"}
  [PSCustomObject]@{Nome="Pedro"; Idade=35; Cidade="MG"}
)
$dados | Export-Csv funcionarios.csv -NoTypeInformation
```

Import-Csv - Importando de CSV

```
# Sintaxe básica
$dados = Import-Csv -Path caminho.csv

# Exemplo 1: Importação básica
$processos = Import-Csv processos.csv

# Os dados são retornados como objetos PSCustomObject
```

```
$processos[0].Name

$processos[0].CPU


# Exemplo 2: Especificar delimitador

$dados = Import-Csv arquivo.csv -Delimiter ';'


# Exemplo 3: Especificar encoding

$dados = Import-Csv arquivo.csv -Encoding UTF8


# Exemplo 4: CSV sem cabeçalho

$dados = Import-Csv sem_cabecalho.csv -Header "Col1", "Col2", "Col3"


# Exemplo 5: Filtrar durante importação

$processosAltoUso = Import-Csv processos.csv | Where-Object {[int]$_.CPU -gt
10}


# Exemplo 6: Converter tipos após importação

$funcionarios = Import-Csv funcionarios.csv | ForEach-Object {
  [PSCustomObject]@{
    Nome = $_.Nome
    Idade = [int]$_.Idade  # Converter para inteiro
    Cidade = $_.Cidade
    Ativo = [bool]$_.Ativo  # Converter para booleano
    Salario = [decimal]$_.Salario  # Converter para decimal
  }
}


# Exemplo 7: Processar em lote
```

```
Import-Csv usuarios.csv | ForEach-Object {
   New-ADUser -Name $_.Nome -GivenName $_.PrimeiroNome -Surname $_.Sobrenome
}
```

ConvertTo-Csv e ConvertFrom-Csv

Diferente de Export/Import, estes convertem para/de formato CSV sem salvar em arquivo:

```
# ConvertTo-Csv: converte objeto para string CSV
$processos = Get-Process | Select-Object -First 3
$csv = $processos | ConvertTo-Csv -NoTypeInformation

# $csv é um array de strings
$csv | ForEach-Object { Write-Host $_ }

# Pode ser útil para enviar por rede, email, etc
$csv | Out-File temp.csv

# ConvertFrom-Csv: converte string CSV para objetos
$textoCSV = @"
Nome,Idade,Cidade
João,30,São Paulo
Maria,25,Rio de Janeiro
Pedro,35,Belo Horizonte
"@

$objetos = $textoCSV | ConvertFrom-Csv
$objetos | Format-Table

# Exemplo prático: ler CSV da web
```

```
$url = "https://exemplo.com/dados.csv"
$dados = (Invoke-WebRequest $url).Content | ConvertFrom-Csv
```

Exemplo Completo: Sistema de Gerenciamento

```
# Sistema simples de cadastro de produtos

# Estrutura do CSV
$arquivoCSV = "produtos.csv"

# Função para adicionar produto
function Add-Produto {
  param(
    [string]$Codigo,
    [string]$Nome,
    [decimal]$Preco,
    [int]$Estoque
  )

  $produto = [PSCustomObject]@{
    Codigo = $Codigo
    Nome = $Nome
    Preco = $Preco
    Estoque = $Estoque
    DataCadastro = Get-Date -Format "dd/MM/yyyy HH:mm:ss"
  }

  $produto | Export-Csv $arquivoCSV -Append -NoTypeInformation
  Write-Host "Produto $Nome cadastrado com sucesso!" -ForegroundColor Green
}
```

```
# Função para listar produtos
function Get-Produtos {
    if (Test-Path $arquivoCSV) {
        $produtos = Import-Csv $arquivoCSV
        $produtos | ForEach-Object {
            [PSCustomObject]@{
                Codigo = $_.Codigo
                Nome = $_.Nome
                Preco = [decimal]$_.Preco
                Estoque = [int]$_.Estoque
                DataCadastro = $_.DataCadastro
            }
        }
    } else {
        Write-Host "Nenhum produto cadastrado." -ForegroundColor Yellow
    }
}

# Função para buscar produto
function Find-Produto {
    param([string]$Codigo)

    $produtos = Get-Produtos
    $produtos | Where-Object Codigo -eq $Codigo
}

# Função para atualizar estoque
```

```
function Update-Estoque {
  param(
    [string]$Codigo,
    [int]$Quantidade
  )

  $produtos = Import-Csv $arquivoCSV | ForEach-Object {
    if ($_.Codigo -eq $Codigo) {
      $_.Estoque = [int]$_.Estoque + $Quantidade
    }
    $_
  }

  $produtos | Export-Csv $arquivoCSV -NoTypeInformation -Force
  Write-Host "Estoque atualizado!" -ForegroundColor Green
}

# Uso
Add-Produto -Codigo "001" -Nome "Mouse" -Preco 45.90 -Estoque 50
Add-Produto -Codigo "002" -Nome "Teclado" -Preco 120.00 -Estoque 30
Get-Produtos | Format-Table -AutoSize
Update-Estoque -Codigo "001" -Quantidade 10
Find-Produto -Codigo "001"
```

3.2.2 Trabalhando com JSON (JavaScript Object Notation)

JSON é um formato leve e amplamente usado para troca de dados, especialmente em APIs REST.

ConvertTo-Json - Convertendo para JSON

 # Sintaxe básica

```
objeto | ConvertTo-Json

# Exemplo 1: Objeto simples
$pessoa = [PSCustomObject]@{
   Nome = "João Silva"
   Idade = 30
   Email = "joao@email.com"
}

$json = $pessoa | ConvertTo-Json
Write-Host $json

# Saída:
# {
#   "Nome": "João Silva",
#   "Idade": 30,
#   "Email": "joao@email.com"
# }

# Exemplo 2: Array de objetos
$funcionarios = @(
   [PSCustomObject]@{Nome="João"; Cargo="Gerente"}
   [PSCustomObject]@{Nome="Maria"; Cargo="Analista"}
)

$funcionarios | ConvertTo-Json

# Exemplo 3: Profundidade (padrão é 2 níveis)
```

```
$objeto = @{
  Nivel1 = @{
    Nivel2 = @{
      Nivel3 = @{
        Dado = "Profundo"
      }
    }
  }
}

# Com profundidade padrão (2)
$objeto | ConvertTo-Json

# Aumentar profundidade
$objeto | ConvertTo-Json -Depth 5

# Exemplo 4: Compacto (sem formatação)
$dados | ConvertTo-Json -Compress

# Exemplo 5: Processos para JSON
Get-Process | Select-Object -First 3 Name, Id, CPU | ConvertTo-Json

# Exemplo 6: Salvar em arquivo
$config = @{
  Servidor = "192.168.1.100"
  Porta = 8080
  SSL = $true
  Timeout = 30
```

```
}

$config | ConvertTo-Json | Out-File config.json -Encoding UTF8

# Exemplo 7: Hashtable aninhada
$configuracao = @{
  Aplicacao = @{
    Nome = "MeuApp"
    Versao = "1.0.0"
  }
  Banco = @{
    Servidor = "localhost"
    Porta = 5432
    Nome = "meubanco"
  }
  Recursos = @("Recurso1", "Recurso2", "Recurso3")
}

$configuracao | ConvertTo-Json -Depth 3 | Out-File app_config.json
```

ConvertFrom-Json - Convertendo de JSON

```
# Sintaxe básica
$objeto = $jsonString | ConvertFrom-Json

# Exemplo 1: Parsing básico
$jsonTexto = '{"Nome":"João","Idade":30,"Cidade":"São Paulo"}'
$objeto = $jsonTexto | ConvertFrom-Json

$objeto.Nome    # "João"
```

```
$objeto.Idade   # 30

# Exemplo 2: Ler de arquivo
$config = Get-Content config.json -Raw | ConvertFrom-Json

$config.Servidor
$config.Porta

# Exemplo 3: Array JSON
$jsonArray = '[
  {"Nome":"João","Idade":30},
  {"Nome":"Maria","Idade":25}
]'

$pessoas = $jsonArray | ConvertFrom-Json
$pessoas.Count      # 2
$pessoas[0].Nome    # "João"

# Exemplo 4: JSON aninhado
$jsonComplexo = @'
{
  "Empresa": "TechCorp",
  "Funcionarios": [
    {
      "Nome": "João",
      "Cargo": "Gerente",
      "Contato": {
        "Email": "joao@tech.com",
```

```
            "Telefone": "11-99999-9999"
        }
    },
    {
        "Nome": "Maria",
        "Cargo": "Analista",
        "Contato": {
            "Email": "maria@tech.com",
            "Telefone": "11-88888-8888"
        }
    }
  ]
}
'@

$dados = $jsonComplexo | ConvertFrom-Json
$dados.Empresa
$dados.Funcionarios[0].Nome
$dados.Funcionarios[0].Contato.Email

# Exemplo 5: Consumir API REST
$url = "https://api.github.com/users/powershell"
$resposta = Invoke-RestMethod -Uri $url  # Já retorna como objeto

# Ou manualmente:
$respostaTexto = Invoke-WebRequest -Uri $url
$dados = $respostaTexto.Content | ConvertFrom-Json
$dados.login
```

```
$dados.public_repos

# Exemplo 6: Modificar e salvar
$config = Get-Content config.json -Raw | ConvertFrom-Json
$config.Porta = 9090
$config.SSL = $false
$config | ConvertTo-Json | Set-Content config.json

# Exemplo 7: Tratar JSON inválido
try {
  $jsonInvalido = '{"Nome":"João","Idade":}'  # JSON malformado
  $objeto = $jsonInvalido | ConvertFrom-Json
}
catch {
  Write-Host "Erro ao processar JSON: $($_.Exception.Message)" -
ForegroundColor Red
}
```

Exemplo Completo: Configuração de Aplicação

```
# Sistema de configuração usando JSON

$arquivoConfig = "app_settings.json"

# Estrutura de configuração padrão
$configPadrao = @{
  Aplicacao = @{
    Nome = "Sistema de Gestão"
    Versao = "2.0.0"
    Ambiente = "Desenvolvimento"
```

```
    }
    Banco = @{
        Servidor = "localhost"
        Porta = 1433
        Nome = "GestaoDb"
        Usuario = "sa"
        Timeout = 30
    }
    Email = @{
        SMTP = "smtp.empresa.com"
        Porta = 587
        SSL = $true
        Remetente = "sistema@empresa.com"
    }
    Logs = @{
        Nivel = "Info"
        Caminho = "C:\Logs\Sistema"
        TamanhoMaximoMB = 100
        RetencaoDias = 30
    }
    Recursos = @(
        "Usuarios",
        "Relatorios",
        "Dashboard",
        "Auditoria"
    )
}
```

```
# Criar configuração se não existir
if (-not (Test-Path $arquivoConfig)) {
  $configPadrao | ConvertTo-Json -Depth 5 | Out-File $arquivoConfig -Encoding UTF8
  Write-Host "Arquivo de configuração criado: $arquivoConfig" -ForegroundColor Green
}

# Ler configuração
function Get-Configuracao {
  if (Test-Path $arquivoConfig) {
    Get-Content $arquivoConfig -Raw | ConvertFrom-Json
  } else {
    Write-Host "Arquivo de configuração não encontrado!" -ForegroundColor Red
  }
}

# Atualizar configuração
function Set-Configuracao {
  param(
    [string]$Secao,
    [string]$Chave,
    $Valor
  )

  $config = Get-Configuracao
  $config.$Secao.$Chave = $Valor
  $config | ConvertTo-Json -Depth 5 | Set-Content $arquivoConfig -Encoding UTF8
```

```
    Write-Host "Configuração atualizada: $Secao.$Chave = $Valor" -
ForegroundColor Green
}

# Exibir configuração formatada
function Show-Configuracao {
    $config = Get-Configuracao

    Write-Host "`n=== CONFIGURAÇÃO DO SISTEMA ===" -ForegroundColor Cyan

    Write-Host "`n[Aplicação]" -ForegroundColor Yellow
    Write-Host "  Nome: $($config.Aplicacao.Nome)"
    Write-Host "  Versão: $($config.Aplicacao.Versao)"
    Write-Host "  Ambiente: $($config.Aplicacao.Ambiente)"

    Write-Host "`n[Banco de Dados]" -ForegroundColor Yellow
    Write-Host "  Servidor: $($config.Banco.Servidor)"
    Write-Host "  Porta: $($config.Banco.Porta)"
    Write-Host "  Database: $($config.Banco.Nome)"

    Write-Host "`n[Email]" -ForegroundColor Yellow
    Write-Host "  SMTP: $($config.Email.SMTP)"
    Write-Host "  Porta: $($config.Email.Porta)"
    Write-Host "  SSL: $($config.Email.SSL)"

    Write-Host "`n[Logs]" -ForegroundColor Yellow
    Write-Host "  Nível: $($config.Logs.Nivel)"
    Write-Host "  Caminho: $($config.Logs.Caminho)"
```

```
    Write-Host "`n[Recursos Habilitados]" -ForegroundColor Yellow
    $config.Recursos | ForEach-Object { Write-Host "  - $_" }
}

# Validar configuração
function Test-Configuracao {
    $config = Get-Configuracao
    $erros = @()

    # Validar banco
    if (-not (Test-Connection $config.Banco.Servidor -Count 1 -Quiet)) {
        $erros += "Servidor de banco inacessível: $($config.Banco.Servidor)"
    }

    # Validar caminho de logs
    if (-not (Test-Path $config.Logs.Caminho)) {
        try {
            New-Item -Path $config.Logs.Caminho -ItemType Directory -Force | Out-Null
            Write-Host "Diretório de logs criado: $($config.Logs.Caminho)" -
ForegroundColor Green
        }
        catch {
            $erros += "Não foi possível criar diretório de logs: $($config.Logs.Caminho)"
        }
    }

    if ($erros.Count -eq 0) {
```

```
        Write-Host "Configuração válida!" -ForegroundColor Green
        return $true
    } else {
        Write-Host "Erros encontrados na configuração:" -ForegroundColor Red
        $erros | ForEach-Object { Write-Host "  - $_" -ForegroundColor Red }
        return $false
    }
}

# Uso
Show-Configuracao
Set-Configuracao -Secao "Banco" -Chave "Porta" -Valor 1434
Test-Configuracao
```

3.2.3 Trabalhando com XML

XML é usado extensivamente no Windows e em configurações de aplicações.

Export-Clixml e Import-Clixml

Serialização completa de objetos PowerShell (preserva tipos):

```
# Export-Clixml: serializa objeto para XML
Get-Process | Export-Clixml processos.xml

# O arquivo XML contém metadados de tipo completos
Get-Content processos.xml | Select-Object -First 20

# Import-Clixml: deserializa de XML
$processos = Import-Clixml processos.xml

# Tipos e propriedades são preservados
$processos[0].GetType()  # System.Diagnostics.Process
```

$processos[0].CPU *# Ainda é um Double*

 # Exemplo prático: salvar estado

$estadoAntes = Get-Service

$estadoAntes | Export-Clixml servicos_antes.xml

 # Fazer mudanças...

Stop-Service wuauserv

 # Comparar depois

$estadoDepois = Get-Service

$estadoAntes = Import-Clixml servicos_antes.xml

Compare-Object $estadoAntes $estadoDepois -Property Name, Status

 # Exemplo: cache de dados

$dadosAPI = Invoke-RestMethod "https://api.exemplo.com/dados"

$dadosAPI | Export-Clixml cache_api.xml

 # Ler do cache (sem chamar API novamente)

$dadosCache = Import-Clixml cache_api.xml

ConvertTo-Xml

Converte objetos para formato XML (não preserva tipos complexos):

 # Sintaxe básica

objeto | ConvertTo-Xml

 # Exemplo 1: Converter processo

$processo = Get-Process | Select-Object -First 1

```
$xml = $processo | ConvertTo-Xml


$xml.GetType()  # System.Xml.XmlDocument


# Exemplo 2: Sem declaração XML e root

$xml = Get-Service | Select-Object -First 3 | ConvertTo-Xml -NoTypeInformation -As String


# Exemplo 3: Profundidade

$dados = @{
  Nivel1 = @{
    Nivel2 = @{
      Valor = "Teste"
    }
  }
}


$xml = $dados | ConvertTo-Xml -Depth 3


# Salvar em arquivo

$xml.Save("C:\temp\dados.xml")
```

Trabalhando com XML Nativo

```
# Criar documento XML

[xml]$xml = @"

<?xml version="1.0" encoding="UTF-8"?>

<Empresa>

  <Nome>TechCorp</Nome>

  <Funcionarios>
```

```
    <Funcionario id="1">
      <Nome>João Silva</Nome>
      <Cargo>Gerente</Cargo>
      <Salario>8000</Salario>
    </Funcionario>
    <Funcionario id="2">
      <Nome>Maria Santos</Nome>
      <Cargo>Analista</Cargo>
      <Salario>5000</Salario>
    </Funcionario>
  </Funcionarios>
</Empresa>
"@

# Acessar elementos (dot notation)
$xml.Empresa.Nome                    # "TechCorp"
$xml.Empresa.Funcionarios.Funcionario.Count   # 2
$xml.Empresa.Funcionarios.Funcionario[0].Nome  # "João Silva"

# Acessar atributos
$xml.Empresa.Funcionarios.Funcionario[0].id   # "1"

# Selecionar com XPath
$xml.SelectNodes("//Funcionario[@id='1']")
$xml.SelectNodes("//Funcionario[Salario>6000]")

# Modificar valores
$xml.Empresa.Funcionarios.Funcionario[0].Salario = "8500"
```

```
# Adicionar novo elemento
$novoFuncionario = $xml.CreateElement("Funcionario")
$novoFuncionario.SetAttribute("id", "3")

$nome = $xml.CreateElement("Nome")
$nome.InnerText = "Pedro Costa"
$novoFuncionario.AppendChild($nome)

$cargo = $xml.CreateElement("Cargo")
$cargo.InnerText = "Desenvolvedor"
$novoFuncionario.AppendChild($cargo)

$xml.Empresa.Funcionarios.AppendChild($novoFuncionario)

# Salvar arquivo
$xml.Save("C:\temp\empresa.xml")

# Ler arquivo XML
[xml]$xmlArquivo = Get-Content "C:\temp\empresa.xml"

# Iterar sobre elementos
foreach ($func in $xml.Empresa.Funcionarios.Funcionario) {
  Write-Host "$($func.Nome) - $($func.Cargo) - R$ $($func.Salario)"
}

# Remover elemento
$funcionarioRemover = $xml.SelectSingleNode("//Funcionario[@id='2']")
```

```
$funcionarioRemover.ParentNode.RemoveChild($funcionarioRemover)
```

Exemplo Completo: Gerenciador de Configuração XML

```
# Sistema de configuração em XML

$arquivoXML = "configuracao.xml"

# Criar configuração padrão
function New-ConfiguracaoXML {
  $xml = [xml]@"
<?xml version="1.0" encoding="UTF-8"?>
<Configuracao>
  <Aplicacao>
    <Nome>Sistema de Gestão</Nome>
    <Versao>2.0.0</Versao>
    <Ambiente>Desenvolvimento</Ambiente>
  </Aplicacao>
  <Conexoes>
    <Banco>
      <Servidor>localhost</Servidor>
      <Porta>1433</Porta>
      <Nome>GestaoDb</Nome>
      <Usuario>sa</Usuario>
    </Banco>
    <Email>
      <SMTP>smtp.empresa.com</SMTP>
      <Porta>587</Porta>
      <SSL>true</SSL>
      <Remetente>sistema@empresa.com</Remetente>
```

```
    </Email>
  </Conexoes>
  <Logs>
    <Nivel>Info</Nivel>
    <Caminho>C:\Logs\Sistema</Caminho>
    <TamanhoMaximoMB>100</TamanhoMaximoMB>
  </Logs>
</Configuracao>
"@
  $xml.Save($arquivoXML)
  Write-Host "Configuração XML criada: $arquivoXML" -ForegroundColor Green
}

# Ler configuração
function Get-ConfiguracaoXML {
  if (Test-Path $arquivoXML) {
    [xml](Get-Content $arquivoXML)
  } else {
    Write-Host "Arquivo não encontrado. Criando novo..." -ForegroundColor Yellow
    New-ConfiguracaoXML
    [xml](Get-Content $arquivoXML)
  }
}

# Obter valor específico
function Get-ValorConfig {
  param(
    [string]$XPath
```

```
    )

    $xml = Get-ConfiguracaoXML
    $node = $xml.SelectSingleNode($XPath)

    if ($node) {
        $node.InnerText
    } else {
        Write-Host "Caminho não encontrado: $XPath" -ForegroundColor Red
    }
}

# Definir valor
function Set-ValorConfig {
    param(
        [string]$XPath,
        [string]$Valor
    )

    $xml = Get-ConfiguracaoXML
    $node = $xml.SelectSingleNode($XPath)

    if ($node) {
        $node.InnerText = $Valor
        $xml.Save($arquivoXML)
        Write-Host "Valor atualizado: $XPath = $Valor" -ForegroundColor Green
    } else {
        Write-Host "Caminho não encontrado: $XPath" -ForegroundColor Red
```

```
    }
}

# Exibir configuração
function Show-ConfiguracaoXML {
    $xml = Get-ConfiguracaoXML

    Write-Host "`n=== CONFIGURAÇÃO XML ===" -ForegroundColor Cyan

    Write-Host "`n[Aplicação]" -ForegroundColor Yellow
    Write-Host "  Nome: $($xml.Configuracao.Aplicacao.Nome)"
    Write-Host "  Versão: $($xml.Configuracao.Aplicacao.Versao)"
    Write-Host "  Ambiente: $($xml.Configuracao.Aplicacao.Ambiente)"

    Write-Host "`n[Banco de Dados]" -ForegroundColor Yellow
    Write-Host "  Servidor: $($xml.Configuracao.Conexoes.Banco.Servidor)"
    Write-Host "  Porta: $($xml.Configuracao.Conexoes.Banco.Porta)"
    Write-Host "  Nome: $($xml.Configuracao.Conexoes.Banco.Nome)"

    Write-Host "`n[Email]" -ForegroundColor Yellow
    Write-Host "  SMTP: $($xml.Configuracao.Conexoes.Email.SMTP)"
    Write-Host "  Porta: $($xml.Configuracao.Conexoes.Email.Porta)"
    Write-Host "  SSL: $($xml.Configuracao.Conexoes.Email.SSL)"

    Write-Host "`n[Logs]" -ForegroundColor Yellow
    Write-Host "  Nível: $($xml.Configuracao.Logs.Nivel)"
    Write-Host "  Caminho: $($xml.Configuracao.Logs.Caminho)"
}
```

 # Uso

New-ConfiguracaoXML

Show-ConfiguracaoXML

Get-ValorConfig -XPath "//Banco/Servidor"

Set-ValorConfig -XPath "//Banco/Porta" -Valor "1434"

3.2.4 Outros Formatos de Dados

Trabalhando com HTML

 # ConvertTo-Html: gera relatórios HTML

$processos = Get-Process | Select-Object -First 10 Name, Id, CPU, WorkingSet

$html = $processos | ConvertTo-Html -Title "Relatório de Processos" -PreContent "<h1>Top 10 Processos</h1>"

$html | Out-File relatorio.html

 # Com CSS customizado

$css = @"

<style>

  body { font-family: Arial, sans-serif; }

  h1 { color: #2E86AB; }

  table { border-collapse: collapse; width: 100%; }

  th { background-color: #2E86AB; color: white; padding: 10px; }

  td { border: 1px solid #ddd; padding: 8px; }

  tr:nth-child(even) { background-color: #f2f2f2; }

</style>

"@

```
$processos | ConvertTo-Html -Head $css -Title "Relatório" -PreContent
"<h1>Processos</h1>" |
  Out-File relatorio_styled.html

# Abrir no browser
Invoke-Item relatorio_styled.html
```

Trabalhando com Arquivos de Texto

```
# Get-Content: ler arquivo
$conteudo = Get-Content arquivo.txt

# Ler linha por linha
Get-Content arquivo.txt | ForEach-Object {
  Write-Host "Linha: $_"
}

# Ler com encoding específico
Get-Content arquivo.txt -Encoding UTF8

# Ler últimas N linhas
Get-Content log.txt -Tail 10

# Ler e monitorar (tail -f)
Get-Content log.txt -Wait -Tail 10

# Set-Content: sobrescrever arquivo
"Nova linha" | Set-Content arquivo.txt

# Add-Content: adicionar ao arquivo
```

"Linha adicional" | Add-Content arquivo.txt

 # Out-File: redirecionar saída

Get-Process | Out-File processos.txt

Trabalhando com Arquivos Binários

 # Ler bytes de arquivo

$bytes = [System.IO.File]::ReadAllBytes("C:\arquivo.exe")

 # Escrever bytes

[System.IO.File]::WriteAllBytes("C:\copia.exe", $bytes)

 # Calcular hash

$hash = Get-FileHash "C:\arquivo.exe" -Algorithm SHA256

$hash.Hash

Conclusão da Seção 3

Nesta seção, exploramos profundamente:

1. **Objetos no PowerShell**: Propriedades, métodos, tipos e como explorá-los
2. **Get-Member**: Ferramenta essencial para descoberta
3. **Criação de objetos customizados**: PSCustomObject e técnicas avançadas
4. **Coleções**: Arrays, ArrayLists, Hashtables e Generic Lists
5. **Conversão e comparação**: Type casting e Compare-Object
6. **CSV**: Importação, exportação e manipulação
7. **JSON**: Trabalho com APIs e configurações modernas
8. **XML**: Manipulação nativa e serialização

Com essas habilidades, você pode manipular dados em diversos formatos e criar soluções robustas de automação e integração.

4. SCRIPTING E AUTOMAÇÃO DE TAREFAS

4.1 Criação de Scripts Simples e Avançados

4.1.1 Fundamentos de Scripts PowerShell

O que é um Script?

Um **script** é um arquivo contendo uma sequência de comandos PowerShell que podem ser executados como uma unidade. Scripts permitem automatizar tarefas repetitivas, criar ferramentas reutilizáveis e implementar lógica complexa.

Características de scripts PowerShell:

- Extensão: .ps1 (PowerShell Script)
- Podem conter cmdlets, funções, variáveis e lógica
- Suportam parâmetros de entrada
- Podem ser assinados digitalmente para segurança
- Executam no contexto do usuário atual

Estrutura Básica de um Script

```
# Cabeçalho com informações do script
<#
.SYNOPSIS
   Breve descrição do que o script faz
.DESCRIPTION
   Descrição detalhada do script
.PARAMETER NomeParametro
   Descrição do parâmetro
.EXAMPLE
   Exemplo de uso do script
.NOTES
   Autor: Seu Nome
   Data: 15/10/2025
   Versão: 1.0
#>
```

```
# Parâmetros do script
param(
  [string]$Parametro1,
  [int]$Parametro2
)

# Configurações iniciais
$ErrorActionPreference = "Stop"

# Corpo do script
# ... código aqui ...

# Saída/resultado
Write-Output "Script concluído"
```

Criando Seu Primeiro Script

Exemplo 1: Script simples (Olá Mundo)

```
# Arquivo: OlaMundo.ps1

# Exibir mensagem
Write-Host "Olá, Mundo!" -ForegroundColor Green

# Obter informações do sistema
Write-Host "`nInformações do Sistema:" -ForegroundColor Cyan
Write-Host "Computador: $env:COMPUTERNAME"
Write-Host "Usuário: $env:USERNAME"
Write-Host "Data: $(Get-Date -Format 'dd/MM/yyyy HH:mm:ss')"
```

 # Pausar antes de fechar

Read-Host "`nPressione Enter para sair"

Executar o script:

 # Navegar até o diretório

cd C:\Scripts

 # Executar

.\OlaMundo.ps1

 # Ou com caminho completo

C:\Scripts\OlaMundo.ps1

Exemplo 2: Script com parâmetros

 # Arquivo: Saudacao.ps1

```
param(
    [string]$Nome = "Visitante"
)

$hora = (Get-Date).Hour

if ($hora -lt 12) {
    $periodo = "Bom dia"
} elseif ($hora -lt 18) {
    $periodo = "Boa tarde"
} else {
    $periodo = "Boa noite"
}
```

```
Write-Host "$periodo, $Nome!" -ForegroundColor Green
Write-Host "Seja bem-vindo ao PowerShell!" -ForegroundColor Cyan
```

Executar com parâmetros:

```
.\Saudacao.ps1
.\Saudacao.ps1 -Nome "João"
.\Saudacao.ps1 -Nome "Maria Silva"
```

4.1.2 Parâmetros e Validação

Declaração de Parâmetros

```
# Sintaxe básica
param(
    [tipo]$NomeParametro
)

# Parâmetros com valores padrão
param(
    [string]$Servidor = "localhost",
    [int]$Porta = 8080,
    [bool]$SSL = $true
)

# Parâmetros obrigatórios
param(
    [Parameter(Mandatory=$true)]
    [string]$Usuario,

    [Parameter(Mandatory=$true)]
    [string]$Senha
)
```

```
# Parâmetros com múltiplos valores
param(
  [string[]]$Computadores,
  [int[]]$Portas
)

# Parâmetros posicionais
param(
  [Parameter(Position=0, Mandatory=$true)]
  [string]$Origem,

  [Parameter(Position=1, Mandatory=$true)]
  [string]$Destino
)
```

Atributos de Validação

```
# ValidateNotNullOrEmpty: não permite nulo ou vazio
param(
  [Parameter(Mandatory=$true)]
  [ValidateNotNullOrEmpty()]
  [string]$Nome
)

# ValidateLength: valida tamanho da string
param(
  [ValidateLength(3, 50)]
  [string]$Usuario
)
```

```
# ValidateRange: valida intervalo numérico
param(
  [ValidateRange(1, 100)]
  [int]$Porcentagem
)

# ValidateSet: valida valores permitidos
param(
  [ValidateSet("Desenvolvimento", "Homologação", "Produção")]
  [string]$Ambiente
)

# ValidatePattern: valida com regex
param(
  [ValidatePattern("^\d{3}\.\d{3}\.\d{3}-\d{2}$")]
  [string]$CPF
)

# ValidateScript: valida com script customizado
param(
  [ValidateScript({Test-Path $_})]
  [string]$Caminho
)

# ValidateCount: valida quantidade de elementos em array
param(
  [ValidateCount(1, 10)]
```

```
    [string[]]$Servidores
)

# AllowNull e AllowEmptyString
param(
    [AllowNull()]
    [string]$Opcional,

    [AllowEmptyString()]
    [string]$PodeSerVazio
)
```

Exemplo Completo: Script com Validação

```
# Arquivo: CriarUsuario.ps1

<#
.SYNOPSIS
    Cria um novo usuário no sistema
.DESCRIPTION
    Script para criação de usuário com validações completas
.PARAMETER Nome
    Nome completo do usuário
.PARAMETER Usuario
    Login do usuário (3-20 caracteres)
.PARAMETER Email
    Email válido do usuário
.PARAMETER Departamento
    Departamento do usuário
.EXAMPLE
```

```
   .\CriarUsuario.ps1 -Nome "João Silva" -Usuario "jsilva" -Email
"joao@empresa.com" -Departamento TI
#>

param(
   [Parameter(Mandatory=$true, HelpMessage="Digite o nome completo")]
   [ValidateNotNullOrEmpty()]
   [ValidateLength(3, 100)]
   [string]$Nome,

   [Parameter(Mandatory=$true)]
   [ValidateLength(3, 20)]
   [ValidatePattern("^[a-zA-Z0-9_-]+$")]
   [string]$Usuario,

   [Parameter(Mandatory=$true)]
   [ValidatePattern("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")]
   [string]$Email,

   [Parameter(Mandatory=$true)]
   [ValidateSet("TI", "RH", "Financeiro", "Operacional", "Comercial")]
   [string]$Departamento,

   [ValidateRange(1, 150)]
   [int]$DiasSenha = 90,

   [switch]$Ativo = $true
)
```

```
# Início do script
Write-Host "`n=== CRIAÇÃO DE USUÁRIO ===" -ForegroundColor Cyan

# Validações adicionais
Write-Host "`nValidando informações..." -ForegroundColor Yellow

# Verificar se usuário já existe (simulação)
$usuariosExistentes = @("admin", "root", "teste")
if ($usuariosExistentes -contains $Usuario.ToLower()) {
    Write-Host "ERRO: Usuário '$Usuario' já existe!" -ForegroundColor Red
    exit 1
}

# Criar objeto de usuário
$novoUsuario = [PSCustomObject]@{
    Nome = $Nome
    Usuario = $Usuario
    Email = $Email
    Departamento = $Departamento
    DataCriacao = Get-Date
    DiasSenha = $DiasSenha
    Ativo = $Ativo
    ID = (Get-Random -Minimum 1000 -Maximum 9999)
}

# Exibir resumo
Write-Host "`nResumo do Usuário:" -ForegroundColor Green
```

```
$novoUsuario | Format-List

# Confirmação
$confirmacao = Read-Host "` nConfirma a criação do usuário? (S/N)"
if ($confirmacao -eq 'S' -or $confirmacao -eq 's') {
  Write-Host "Usuário criado com sucesso!" -ForegroundColor Green

  # Salvar em arquivo (simulação)
  $novoUsuario | Export-Csv "usuarios.csv" -Append -NoTypeInformation

  # Log
  $logEntry = "$(Get-Date -Format 'yyyy-MM-dd HH:mm:ss') - Usuário $Usuario criado"
  $logEntry | Out-File "usuarios.log" -Append
} else {
  Write-Host "Operação cancelada." -ForegroundColor Yellow
}
```

4.1.3 Funções em Scripts

Criando Funções

```
# Sintaxe básica
function Get-MeuDado {
  # código da função
}

# Função com parâmetros
function Get-Saudacao {
  param(
    [string]$Nome
```

```
    )

    return "Olá, $Nome!"
}

# Função avançada (com cmdlet binding)
function Get-InformacaoServidor {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$true, ValueFromPipeline=$true)]
        [string]$ComputerName
    )

    begin {
        Write-Verbose "Iniciando coleta de informações"
    }

    process {
        try {
            $os = Get-CimInstance -ClassName Win32_OperatingSystem -
ComputerName $ComputerName

            [PSCustomObject]@{
                Computador = $ComputerName
                SO = $os.Caption
                Versao = $os.Version
                Arquitetura = $os.OSArchitecture
                MemoriaGB = [math]::Round($os.TotalVisibleMemorySize / 1MB, 2)
```

```
        }
    }
    catch {
        Write-Error "Erro ao coletar dados de $ComputerName : $_"
    }
  }

  end {
    Write-Verbose "Coleta finalizada"
  }
}

# Usar a função
Get-InformacaoServidor -ComputerName "localhost" -Verbose
```

Escopo de Funções

```
# Função em escopo de script
function Script:MinhaFuncao {
  Write-Host "Função de script"
}

# Função em escopo global
function Global:OutraFuncao {
  Write-Host "Função global"
}

# Função privada (não exportada de módulos)
function Private:FuncaoPrivada {
  Write-Host "Função privada"
```

```
}
```

Exemplo Completo: Biblioteca de Funções

```
# Arquivo: BibliotecaUtils.ps1

<#
.SYNOPSIS
    Biblioteca de funções utilitárias
#>

# Função para validar email
function Test-Email {
    <#
    .SYNOPSIS
        Valida formato de email
    .PARAMETER Email
        Email a ser validado
    .EXAMPLE
        Test-Email -Email "usuario@dominio.com"
    #>
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$true)]
        [string]$Email
    )

    $pattern = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return $Email -match $pattern
}
```

```powershell
# Função para obter tamanho de diretório
function Get-DirectorySize {
  <#
  .SYNOPSIS
    Calcula tamanho total de um diretório
  .PARAMETER Path
    Caminho do diretório
  .EXAMPLE
    Get-DirectorySize -Path "C:\Logs"
  #>
  [CmdletBinding()]
  param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$Path
  )

  $size = (Get-ChildItem -Path $Path -Recurse -File |
        Measure-Object -Property Length -Sum).Sum

  [PSCustomObject]@{
    Caminho = $Path
    TotalBytes = $size
    TotalMB = [math]::Round($size / 1MB, 2)
    TotalGB = [math]::Round($size / 1GB, 2)
  }
}
```

```
# Função para criar backup
function New-Backup {
  <#
  .SYNOPSIS
    Cria backup compactado de um diretório
  .PARAMETER Source
    Diretório de origem
  .PARAMETER Destination
    Diretório de destino
  .EXAMPLE
    New-Backup -Source "C:\Dados" -Destination "D:\Backups"
  #>
  [CmdletBinding()]
  param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$Source,

    [Parameter(Mandatory=$true)]
    [string]$Destination
  )

  # Criar diretório de destino se não existir
  if (-not (Test-Path $Destination)) {
    New-Item -Path $Destination -ItemType Directory | Out-Null
  }
```

```
    # Nome do arquivo de backup
    $timestamp = Get-Date -Format "yyyyMMdd_HHmmss"
    $backupName = "Backup_$timestamp.zip"
    $backupPath = Join-Path $Destination $backupName

    Write-Host "Criando backup: $backupPath" -ForegroundColor Yellow

    try {
        # Criar arquivo ZIP
        Compress-Archive -Path $Source -DestinationPath $backupPath -
CompressionLevel Optimal

        $fileInfo = Get-Item $backupPath

        Write-Host "Backup criado com sucesso!" -ForegroundColor Green
        Write-Host "Tamanho: $([math]::Round($fileInfo.Length / 1MB, 2)) MB" -
ForegroundColor Cyan

        return [PSCustomObject]@{
            Sucesso = $true
            ArquivoBackup = $backupPath
            Tamanho = $fileInfo.Length
            DataCriacao = $fileInfo.CreationTime
        }
    }
    catch {
        Write-Host "Erro ao criar backup: $_" -ForegroundColor Red
        return [PSCustomObject]@{
            Sucesso = $false
```

```
            Erro = $_.Exception.Message
        }
    }
}

# Função para limpeza de arquivos antigos
function Remove-OldFiles {
    <#
    .SYNOPSIS
        Remove arquivos mais antigos que X dias
    .PARAMETER Path
        Diretório a ser limpo
    .PARAMETER Days
        Arquivos mais antigos que esta quantidade de dias serão removidos
    .PARAMETER WhatIf
        Simula a operação sem executar
    .EXAMPLE
        Remove-OldFiles -Path "C:\Logs" -Days 30
    #>
    [CmdletBinding(SupportsShouldProcess=$true)]
    param(
        [Parameter(Mandatory=$true)]
        [ValidateScript({Test-Path $_})]
        [string]$Path,

        [Parameter(Mandatory=$true)]
        [ValidateRange(1, 365)]
        [int]$Days
```

```
    )

    $dataLimite = (Get-Date).AddDays(-$Days)

    Write-Host "` nBuscando arquivos anteriores a: $($dataLimite.ToString('dd/MM/yyyy'))" -ForegroundColor Yellow

    $arquivosAntigos = Get-ChildItem -Path $Path -File -Recurse |
                Where-Object { $_.LastWriteTime -lt $dataLimite }

    if ($arquivosAntigos.Count -eq 0) {
        Write-Host "Nenhum arquivo antigo encontrado." -ForegroundColor Green
        return
    }

    Write-Host "Encontrados $($arquivosAntigos.Count) arquivos antigos" -ForegroundColor Cyan

    $tamanhoTotal = ($arquivosAntigos | Measure-Object -Property Length -Sum).Sum
    Write-Host "Espaço a ser liberado: $([math]::Round($tamanhoTotal / 1MB, 2)) MB" -ForegroundColor Cyan

    $removidos = 0
    foreach ($arquivo in $arquivosAntigos) {
        if ($PSCmdlet.ShouldProcess($arquivo.FullName, "Remover arquivo")) {
            try {
                Remove-Item -Path $arquivo.FullName -Force
                $removidos++
```

```
            Write-Verbose "Removido: $($arquivo.Name)"
        }
        catch {
            Write-Warning "Erro ao remover $($arquivo.Name): $_"
        }
    }
  }

  Write-Host "`n$removidos arquivos removidos com sucesso!" -ForegroundColor Green
}

# Função para monitorar uso de disco
function Get-DiskUsage {
  <#
  .SYNOPSIS
    Retorna uso de disco com alertas
  .PARAMETER ThresholdPercent
    Percentual de alerta
  .EXAMPLE
    Get-DiskUsage -ThresholdPercent 80
  #>
  [CmdletBinding()]
  param(
    [ValidateRange(1, 100)]
    [int]$ThresholdPercent = 80
  )
```

```
    $discos = Get-CimInstance -ClassName Win32_LogicalDisk -Filter "DriveType=3"

    foreach ($disco in $discos) {
        $percentualUsado = [math]::Round((($disco.Size - $disco.FreeSpace) / $disco.Size) * 100, 2)
        $status = if ($percentualUsado -ge $ThresholdPercent) { " ⚠️ ALERTA" } else { " ✅ OK" }

        [PSCustomObject]@{
            Disco = $disco.DeviceID
            TamanhoGB = [math]::Round($disco.Size / 1GB, 2)
            LivreGB = [math]::Round($disco.FreeSpace / 1GB, 2)
            UsadoGB = [math]::Round(($disco.Size - $disco.FreeSpace) / 1GB, 2)
            PercentualUsado = $percentualUsado
            Status = $status
        }
    }
}

# Exportar funções (se usado como módulo)
Export-ModuleMember -Function Test-Email, Get-DirectorySize, New-Backup, Remove-OldFiles, Get-DiskUsage
```

Usar a biblioteca:

```
# Dot source (carregar funções no escopo atual)
. .\BibliotecaUtils.ps1

# Usar funções
Test-Email -Email "usuario@exemplo.com"
Get-DirectorySize -Path "C:\Logs"
```

```
New-Backup -Source "C:\Dados" -Destination "D:\Backups"

Remove-OldFiles -Path "C:\Logs" -Days 30 -WhatIf

Get-DiskUsage -ThresholdPercent 75
```

4.1.4 Tratamento de Erros

Try-Catch-Finally

```
# Estrutura básica
try {
    # Código que pode gerar erro
    Get-Item "C:\ArquivoInexistente.txt" -ErrorAction Stop
}
catch {
    # Tratar erro
    Write-Host "Erro capturado: $($_.Exception.Message)" -ForegroundColor Red
}
finally {
    # Sempre executado (opcional)
    Write-Host "Bloco finally executado"
}


# Múltiplos catches (tipos específicos)
try {
    $resultado = 10 / 0
}
catch [System.DivideByZeroException] {
    Write-Host "Erro: Divisão por zero"
}
catch [System.IO.FileNotFoundException] {
    Write-Host "Erro: Arquivo não encontrado"
```

```
}
catch {
  Write-Host "Erro genérico: $_"
}

# Variável automática $_ ou $PSItem
try {
  Get-Content "arquivo_inexistente.txt" -ErrorAction Stop
}
catch {
  Write-Host "Mensagem: $($_.Exception.Message)"
  Write-Host "Tipo: $($_.Exception.GetType().FullName)"
  Write-Host "Linha: $($_.InvocationInfo.ScriptLineNumber)"
  Write-Host "Comando: $($_.InvocationInfo.Line)"
}
```

ErrorAction e ErrorActionPreference

```
# ErrorActionPreference (escopo do script)
$ErrorActionPreference = "Stop"       # Trata erros como terminantes
$ErrorActionPreference = "Continue"   # Padrão - exibe erro e continua
$ErrorActionPreference = "SilentlyContinue"  # Suprime erro
$ErrorActionPreference = "Inquire"    # Pergunta ao usuário

# ErrorAction (por comando)
Get-Item "arquivo.txt" -ErrorAction Stop
Get-Item "arquivo.txt" -ErrorAction SilentlyContinue
Get-Item "arquivo.txt" -ErrorAction Ignore

# Capturar erro em variável
```

```
Get-Item "arquivo.txt" -ErrorAction SilentlyContinue -ErrorVariable meuErro
if ($meuErro) {
  Write-Host "Erro ocorreu: $meuErro"
}

# Variável automática $Error
$Error[0]      # Último erro
$Error.Count    # Quantidade de erros
$Error.Clear()  # Limpar histórico de erros
```

Throw - Gerar Erros

```
# Throw simples
throw "Erro customizado"

# Throw com tipo
throw [System.ArgumentException]::new("Parâmetro inválido")

# Throw condicional
function Divide-Numero {
  param([int]$Numero, [int]$Divisor)

  if ($Divisor -eq 0) {
    throw "Divisor não pode ser zero"
  }

  return $Numero / $Divisor
}

try {
```

```
    Divide-Numero -Numero 10 -Divisor 0
}
catch {
    Write-Host "Erro: $_" -ForegroundColor Red
}
```

Exemplo Completo: Script com Tratamento Robusto de Erros

```
# Arquivo: ProcessarArquivos.ps1

<#
.SYNOPSIS
    Processa arquivos com tratamento robusto de erros
#>

param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$OrigemPath,

    [Parameter(Mandatory=$true)]
    [string]$DestinoPath,

    [switch]$ContinueOnError
)

# Configuração de erro
$ErrorActionPreference = if ($ContinueOnError) { "Continue" } else { "Stop" }

# Contador de resultados
```

```
$script:sucessos = 0
$script:falhas = 0
$script:erros = @()

# Função para registrar erro
function Write-ErrorLog {
    param(
        [string]$Arquivo,
        [string]$Mensagem
    )

    $script:falhas++
    $erro = [PSCustomObject]@{
        Timestamp = Get-Date
        Arquivo = $Arquivo
        Erro = $Mensagem
    }
    $script:erros += $erro

    Write-Host "ERRO: $Arquivo - $Mensagem" -ForegroundColor Red
}

# Função para processar arquivo
function Process-File {
    param([System.IO.FileInfo]$Arquivo)

    try {
        Write-Verbose "Processando: $($Arquivo.Name)"
```

```
    # Validar se arquivo não está em uso
    try {
      $stream = [System.IO.File]::Open($Arquivo.FullName, 'Open', 'Read', 'None')
      $stream.Close()
      $stream.Dispose()
    }
    catch {
      throw "Arquivo em uso por outro processo"
    }

    # Validar tamanho
    if ($Arquivo.Length -eq 0) {
      throw "Arquivo vazio"
    }

    # Criar diretório de destino
    if (-not (Test-Path $DestinoPath)) {
      New-Item -Path $DestinoPath -ItemType Directory -Force | Out-Null
    }

    # Copiar arquivo
    $destino = Join-Path $DestinoPath $Arquivo.Name
    Copy-Item -Path $Arquivo.FullName -Destination $destino -Force -ErrorAction
Stop

    # Validar cópia
    $arquivoDestino = Get-Item $destino
```

```
        if ($arquivoDestino.Length -ne $Arquivo.Length) {
            throw "Tamanho do arquivo de destino difere do original"
        }

        $script:sucessos++
        Write-Host "✓ $($Arquivo.Name) processado com sucesso" -ForegroundColor Green

        return $true
    }
    catch {
        Write-ErrorLog -Arquivo $Arquivo.Name -Mensagem $_.Exception.Message
        return $false
    }
}

# Início do processamento
Write-Host "`n=== PROCESSAMENTO DE ARQUIVOS ===" -ForegroundColor Cyan
Write-Host "Origem: $OrigemPath" -ForegroundColor Yellow
Write-Host "Destino: $DestinoPath" -ForegroundColor Yellow

try {
    # Obter arquivos
    Write-Host "`nBuscando arquivos..." -ForegroundColor Yellow
    $arquivos = Get-ChildItem -Path $OrigemPath -File -ErrorAction Stop

    if ($arquivos.Count -eq 0) {
        Write-Host "Nenhum arquivo encontrado no diretório de origem." -ForegroundColor Yellow
```

```
        exit 0
    }

    Write-Host "Encontrados $($arquivos.Count) arquivos" -ForegroundColor Cyan

    # Processar cada arquivo
    Write-Host "`nProcessando arquivos..." -ForegroundColor Yellow

    foreach ($arquivo in $arquivos) {
        $resultado = Process-File -Arquivo $arquivo

        # Se não for para continuar em erro e houve falha, parar
        if (-not $ContinueOnError -and -not $resultado) {
            throw "Processamento interrompido devido a erro"
        }
    }
}
catch {
    Write-Host "`nERRO CRÍTICO: $($_.Exception.Message)" -ForegroundColor Red
    Write-Host "StackTrace: $($_.ScriptStackTrace)" -ForegroundColor DarkRed
    exit 1
}
finally {
    # Relatório final
    Write-Host "`n=== RESULTADO DO PROCESSAMENTO ===" -ForegroundColor
Cyan
    Write-Host "Total de arquivos: $($arquivos.Count)" -ForegroundColor White
    Write-Host "Sucessos: $script:sucessos" -ForegroundColor Green
```

```
    Write-Host "Falhas: $script:falhas" -ForegroundColor Red

    # Salvar log de erros se houver
    if ($script:erros.Count -gt 0) {
        $logPath = "ProcessamentoErros_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv"
        $script:erros | Export-Csv $logPath -NoTypeInformation
        Write-Host "`nLog de erros salvo em: $logPath" -ForegroundColor Yellow
    }

    # Status de saída
    if ($script:falhas -eq 0) {
        Write-Host "`nProcessamento concluído com sucesso!" -ForegroundColor Green
        exit 0
    } else {
        Write-Host "`nProcessamento concluído com erros." -ForegroundColor Yellow
        exit 1
    }
}
```

4.1.5 Logging e Debugging

Write-Host vs Write-Output vs Write-Verbose

```
# Write-Host: saída visual (não vai para pipeline)
Write-Host "Mensagem para o usuário" -ForegroundColor Green

# Write-Output: saída para pipeline
Write-Output "Dados para processamento"

# Write-Verbose: mensagens detalhadas (precisa -Verbose)
```

```
Write-Verbose "Informação de debug"

# Write-Warning: avisos
Write-Warning "Isso pode causar problemas"

# Write-Error: erros não terminantes
Write-Error "Ocorreu um erro"

# Write-Information: informações (PS 5.0+)
Write-Information "Informação importante" -InformationAction Continue

# Write-Debug: mensagens de debug
Write-Debug "Valor da variável: $valor"
```

Sistema de Logging Completo

```
# Arquivo: LoggingSystem.ps1

# Enum para níveis de log
enum LogLevel {
    DEBUG = 0
    INFO = 1
    WARNING = 2
    ERROR = 3
    CRITICAL = 4
}

# Configuração de logging
$script:LogConfig = @{
    LogPath = ".\logs"
```

```
    LogFile = "aplicacao_$(Get-Date -Format 'yyyyMMdd').log"
    MinLevel = [LogLevel]::INFO
    ConsoleOutput = $true
    FileOutput = $true
}

# Função principal de logging
function Write-Log {
    [CmdletBinding()]
    param(
        [Parameter(Mandatory=$true)]
        [string]$Message,

        [LogLevel]$Level = [LogLevel]::INFO,

        [string]$Source = "SYSTEM"
    )

    # Verificar nível mínimo
    if ($Level -lt $script:LogConfig.MinLevel) {
        return
    }

    # Formato da mensagem
    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss.fff"
    $logEntry = "[$timestamp] [$Level] [$Source] $Message"

    # Saída no console
```

```
    if ($script:LogConfig.ConsoleOutput) {
        $color = switch ($Level) {
            ([LogLevel]::DEBUG)    { "Gray" }
            ([LogLevel]::INFO)     { "White" }
            ([LogLevel]::WARNING)  { "Yellow" }
            ([LogLevel]::ERROR)    { "Red" }
            ([LogLevel]::CRITICAL) { "Magenta" }
        }
        Write-Host $logEntry -ForegroundColor $color
    }

    # Saída em arquivo
    if ($script:LogConfig.FileOutput) {
        # Criar diretório se não existir
        if (-not (Test-Path $script:LogConfig.LogPath)) {
            New-Item -Path $script:LogConfig.LogPath -ItemType Directory -Force | Out-
Null
        }

        $logFilePath = Join-Path $script:LogConfig.LogPath $script:LogConfig.LogFile
        $logEntry | Out-File -FilePath $logFilePath -Append -Encoding UTF8
    }
}

# Funções auxiliares
function Write-LogDebug {
    param([string]$Message, [string]$Source = "DEBUG")
    Write-Log -Message $Message -Level ([LogLevel]::DEBUG) -Source $Source
```

```
}

function Write-LogInfo {
    param([string]$Message, [string]$Source = "INFO")
    Write-Log -Message $Message -Level ([LogLevel]::INFO) -Source $Source
}

function Write-LogWarning {
    param([string]$Message, [string]$Source = "WARNING")
    Write-Log -Message $Message -Level ([LogLevel]::WARNING) -Source $Source
}

function Write-LogError {
    param([string]$Message, [string]$Source = "ERROR")
    Write-Log -Message $Message -Level ([LogLevel]::ERROR) -Source $Source
}

function Write-LogCritical {
    param([string]$Message, [string]$Source = "CRITICAL")
    Write-Log -Message $Message -Level ([LogLevel]::CRITICAL) -Source $Source
}

# Função para configurar logging
function Set-LogConfiguration {
    param(
        [string]$LogPath,
        [LogLevel]$MinLevel,
        [bool]$ConsoleOutput,
```

```
    [bool]$FileOutput
  )

  if ($LogPath) { $script:LogConfig.LogPath = $LogPath }
  if ($MinLevel) { $script:LogConfig.MinLevel = $MinLevel }
  if ($PSBoundParameters.ContainsKey('ConsoleOutput')) {
    $script:LogConfig.ConsoleOutput = $ConsoleOutput
  }
  if ($PSBoundParameters.ContainsKey('FileOutput')) {
    $script:LogConfig.FileOutput = $FileOutput
  }
}

# Função para limpar logs antigos
function Clear-OldLogs {
  param(
    [int]$DaysToKeep = 30
  )

  $dataLimite = (Get-Date).AddDays(-$DaysToKeep)
  $arquivosAntigos = Get-ChildItem -Path $script:LogConfig.LogPath -Filter "*.log"
|
          Where-Object { $_.LastWriteTime -lt $dataLimite }

  foreach ($arquivo in $arquivosAntigos) {
    Remove-Item -Path $arquivo.FullName -Force
    Write-LogInfo "Log antigo removido: $($arquivo.Name)"
  }
```

}

 # Exemplo de uso

Write-LogInfo "Aplicação iniciada"

Write-LogDebug "Variável X = 10"

Write-LogWarning "Conexão lenta detectada"

Write-LogError "Falha ao conectar ao servidor"

Write-LogCritical "Sistema de pagamento indisponível"

Debugging com Set-PSDebug

 # Ativar debug (mostra cada linha executada)

Set-PSDebug -Trace 1

 # Nível de trace mais detalhado

Set-PSDebug -Trace 2

 # Strict mode (variáveis não inicializadas geram erro)

Set-PSDebug -Strict

 # Desativar debug

Set-PSDebug -Off

 # Exemplo

Set-PSDebug -Trace 1

$nome = "João"

Write-Host "Olá, $nome"

Set-PSDebug -Off

Breakpoints e Debugging Interativo

 # Definir breakpoint em linha específica

```
Set-PSBreakpoint -Script "MeuScript.ps1" -Line 10

# Breakpoint em variável (quando modificada)
Set-PSBreakpoint -Script "MeuScript.ps1" -Variable "contador"

# Breakpoint em comando
Set-PSBreakpoint -Script "MeuScript.ps1" -Command "Get-Process"

# Listar breakpoints
Get-PSBreakpoint

# Remover breakpoint
Remove-PSBreakpoint -Id 1

# Remover todos
Get-PSBreakpoint | Remove-PSBreakpoint

# No VS Code:
# - F9: Toggle breakpoint
# - F5: Iniciar debug
# - F10: Step Over
# - F11: Step Into
# - Shift+F11: Step Out
```

4.1.6 Scripts Avançados - Exemplos Práticos

Exemplo 1: Script de Monitoramento de Servidor

```
# Arquivo: MonitorServidor.ps1

<#
```

```
<#
.SYNOPSIS
    Monitora recursos do servidor e gera alertas
.DESCRIPTION
    Script completo de monitoramento com múltiplas verificações
.PARAMETER ComputerName
    Nome do servidor a monitorar
.PARAMETER AlertEmail
    Email para envio de alertas
.PARAMETER CPUThreshold
    Limite de uso de CPU (%)
.PARAMETER MemoryThreshold
    Limite de uso de memória (%)
.PARAMETER DiskThreshold
    Limite de uso de disco (%)
.EXAMPLE
    .\MonitorServidor.ps1 -ComputerName "SERVER01" -AlertEmail "admin@empresa.com"
#>

[CmdletBinding()]
param(
    [Parameter(Mandatory=$false)]
    [string]$ComputerName = $env:COMPUTERNAME,

    [Parameter(Mandatory=$false)]
    [ValidatePattern("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")]
    [string]$AlertEmail,
```

```
    [ValidateRange(1, 100)]
    [int]$CPUThreshold = 80,

    [ValidateRange(1, 100)]
    [int]$MemoryThreshold = 85,

    [ValidateRange(1, 100)]
    [int]$DiskThreshold = 90,

    [switch]$ContinuousMonitoring,

    [int]$IntervalSeconds = 60
)

# Configurar logging
$logPath = ".\MonitorLogs"
if (-not (Test-Path $logPath)) {
    New-Item -Path $logPath -ItemType Directory | Out-Null
}

$logFile = Join-Path $logPath "Monitor_$(Get-Date -Format 'yyyyMMdd').log"

function Write-MonitorLog {
    param(
        [string]$Message,
        [string]$Level = "INFO"
    )
```

```
    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    $logEntry = "[$timestamp] [$Level] $Message"

    $color = switch ($Level) {
        "INFO"    { "White" }
        "WARNING" { "Yellow" }
        "ERROR"   { "Red" }
        "ALERT"   { "Magenta" }
        default   { "Gray" }
    }

    Write-Host $logEntry -ForegroundColor $color
    $logEntry | Out-File -FilePath $logFile -Append -Encoding UTF8
}

function Get-CPUUsage {
    try {
        $cpu = Get-CimInstance -ClassName Win32_Processor -ComputerName
$ComputerName
        $usage = $cpu.LoadPercentage

        return [PSCustomObject]@{
            Metric = "CPU"
            Value = $usage
            Threshold = $CPUThreshold
            Status = if ($usage -ge $CPUThreshold) { "ALERT" } else { "OK" }
            Message = "Uso de CPU: $usage%"
        }
```

```
    }
    catch {
        Write-MonitorLog "Erro ao obter uso de CPU: $_" -Level "ERROR"
        return $null
    }
}

function Get-MemoryUsage {
    try {
        $os = Get-CimInstance -ClassName Win32_OperatingSystem -
ComputerName $ComputerName
        $totalMemory = $os.TotalVisibleMemorySize
        $freeMemory = $os.FreePhysicalMemory
        $usedPercent = [math]::Round((($totalMemory - $freeMemory) /
$totalMemory) * 100, 2)

        return [PSCustomObject]@{
            Metric = "Memory"
            Value = $usedPercent
            Threshold = $MemoryThreshold
            Status = if ($usedPercent -ge $MemoryThreshold) { "ALERT" } else { "OK" }
            Message = "Uso de Memória: $usedPercent% (Livre:
$([math]::Round($freeMemory/1MB, 2)) GB)"
        }
    }
    catch {
        Write-MonitorLog "Erro ao obter uso de memória: $_" -Level "ERROR"
        return $null
    }
```

```
}

function Get-DiskUsage {
  try {
    $discos = Get-CimInstance -ClassName Win32_LogicalDisk -Filter "DriveType=3" -ComputerName $ComputerName
    $resultados = @()

    foreach ($disco in $discos) {
      $usedPercent = [math]::Round((($disco.Size - $disco.FreeSpace) / $disco.Size) * 100, 2)

      $resultados += [PSCustomObject]@{
        Metric = "Disk $($disco.DeviceID)"
        Value = $usedPercent
        Threshold = $DiskThreshold
        Status = if ($usedPercent -ge $DiskThreshold) { "ALERT" } else { "OK" }
        Message = "Disco $($disco.DeviceID): $usedPercent% usado (Livre: $([math]::Round($disco.FreeSpace/1GB, 2)) GB)"
      }
    }

    return $resultados
  }
  catch {
    Write-MonitorLog "Erro ao obter uso de disco: $_" -Level "ERROR"
    return $null
  }
}
```

```
function Get-ServiceStatus {
  try {
    $servicosCriticos = @("wuauserv", "BITS", "EventLog", "WinRM")
    $resultados = @()

    foreach ($servico in $servicosCriticos) {
      $svc = Get-Service -Name $servico -ComputerName $ComputerName -
ErrorAction SilentlyContinue

      if ($svc) {
        $resultados += [PSCustomObject]@{
          Metric = "Service $servico"
          Value = $svc.Status
          Threshold = "Running"
          Status = if ($svc.Status -ne "Running") { "ALERT" } else { "OK" }
          Message = "Serviço $($svc.DisplayName): $($svc.Status)"
        }
      }
    }

    return $resultados
  }
  catch {
    Write-MonitorLog "Erro ao verificar serviços: $_" -Level "ERROR"
    return $null
  }
}
```

```
function Send-AlertEmail {
    param(
        [array]$Alerts
    )

    if (-not $AlertEmail) {
        return
    }

    try {
        $body = "ALERTAS DE MONITORAMENTO - $ComputerName`n`n"
        $body += "Data/Hora: $(Get-Date -Format 'dd/MM/yyyy HH:mm:ss')`n`n"
        $body += "Alertas:`n"

        foreach ($alert in $Alerts) {
            $body += "- $($alert.Message)`n"
        }

        # Configurar parâmetros de email (ajustar conforme sua infraestrutura)
        $emailParams = @{
            From = "monitor@empresa.com"
            To = $AlertEmail
            Subject = "ALERTA: $ComputerName - $(Get-Date -Format 'dd/MM/yyyy
HH:mm:ss')"
            Body = $body
            SmtpServer = "smtp.empresa.com"
        }
```

```
        Send-MailMessage @emailParams
        Write-MonitorLog "Email de alerta enviado para $AlertEmail" -Level "INFO"
    }
    catch {
        Write-MonitorLog "Erro ao enviar email: $_" -Level "ERROR"
    }
}

function Start-Monitoring {
    Write-MonitorLog "===== MONITORAMENTO INICIADO =====" -Level "INFO"
    Write-MonitorLog "Servidor: $ComputerName" -Level "INFO"
    Write-MonitorLog "Limites: CPU=$CPUThreshold%,
Memória=$MemoryThreshold%, Disco=$DiskThreshold%" -Level "INFO"

    do {
        Write-MonitorLog "`n----- Nova Verificação -----" -Level "INFO"

        $todasMetricas = @()
        $alertas = @()

        # Coletar métricas
        $cpu = Get-CPUUsage
        if ($cpu) {
            $todasMetricas += $cpu
            if ($cpu.Status -eq "ALERT") { $alertas += $cpu }
        }
```

```
    $memoria = Get-MemoryUsage
    if ($memoria) {
        $todasMetricas += $memoria
        if ($memoria.Status -eq "ALERT") { $alertas += $memoria }
    }

    $discos = Get-DiskUsage
    if ($discos) {
        $todasMetricas += $discos
        $alertas += $discos | Where-Object Status -eq "ALERT"
    }

    $servicos = Get-ServiceStatus
    if ($servicos) {
        $todasMetricas += $servicos
        $alertas += $servicos | Where-Object Status -eq "ALERT"
    }

    # Exibir resultados
    foreach ($metrica in $todasMetricas) {
        $level = if ($metrica.Status -eq "ALERT") { "ALERT" } else { "INFO" }
        Write-MonitorLog $metrica.Message -Level $level
    }

    # Enviar alertas se houver
    if ($alertas.Count -gt 0) {
        Write-MonitorLog "`n!!! $($alertas.Count) ALERTA(S) DETECTADO(S) !!!" -
Level "ALERT"
```

```
            Send-AlertEmail -Alerts $alertas
        }

        # Salvar relatório
        $relatorioPath = Join-Path $logPath "Relatorio_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv"
        $todasMetricas | Export-Csv -Path $relatorioPath -NoTypeInformation

        if ($ContinuousMonitoring) {
            Write-MonitorLog "`nPróxima verificação em $IntervalSeconds segundos..." -Level "INFO"
            Start-Sleep -Seconds $IntervalSeconds
        }

    } while ($ContinuousMonitoring)

    Write-MonitorLog "`n===== MONITORAMENTO ENCERRADO =====" -Level "INFO"
}

# Executar monitoramento
try {
    Start-Monitoring
}
catch {
    Write-MonitorLog "ERRO CRÍTICO: $($_.Exception.Message)" -Level "ERROR"
    Write-MonitorLog "StackTrace: $($_.ScriptStackTrace)" -Level "ERROR"
    exit 1
}
```

Uso do script:

 # Monitoramento único

.\MonitorServidor.ps1 -ComputerName "SERVER01"

 # Monitoramento contínuo

.\MonitorServidor.ps1 -ContinuousMonitoring -IntervalSeconds 300

 # Com alertas por email

.\MonitorServidor.ps1 -AlertEmail "admin@empresa.com" -CPUThreshold 75

 # Monitoramento contínuo com alertas

.\MonitorServidor.ps1 -ContinuousMonitoring -IntervalSeconds 60 -AlertEmail "admin@empresa.com"

4.2 Gerenciamento de Tarefas Agendadas

4.2.1 Introdução às Tarefas Agendadas

O que são Tarefas Agendadas?

Tarefas agendadas (Scheduled Tasks) permitem executar scripts, programas ou comandos automaticamente em horários específicos ou em resposta a eventos do sistema.

Casos de uso comuns:

- Backups automáticos
- Limpeza de arquivos temporários
- Coleta de métricas de sistema
- Sincronização de dados
- Envio de relatórios
- Manutenção de sistemas
- Monitoramento contínuo

Componentes principais:

- **Trigger (Gatilho)**: Quando a tarefa deve executar

- **Action (Ação)**: O que a tarefa deve fazer
- **Conditions (Condições)**: Requisitos para execução
- **Settings (Configurações)**: Comportamento da tarefa

4.2.2 Cmdlets para Gerenciamento de Tarefas

PowerShell oferece cmdlets nativos para gerenciar tarefas agendadas:

```
# Cmdlets principais
Get-ScheduledTask       # Listar tarefas
New-ScheduledTask       # Criar nova tarefa
Register-ScheduledTask  # Registrar tarefa no sistema
Set-ScheduledTask       # Modificar tarefa existente
Unregister-ScheduledTask  # Remover tarefa
Start-ScheduledTask     # Executar tarefa manualmente
Stop-ScheduledTask      # Parar execução
Enable-ScheduledTask    # Habilitar tarefa
Disable-ScheduledTask   # Desabilitar tarefa
Get-ScheduledTaskInfo   # Informações de execução
```

Listar Tarefas Agendadas

```
# Listar todas as tarefas
Get-ScheduledTask

# Listar tarefas ativas
Get-ScheduledTask | Where-Object State -eq 'Ready'

# Listar tarefas desabilitadas
Get-ScheduledTask | Where-Object State -eq 'Disabled'

# Buscar tarefa específica
Get-ScheduledTask -TaskName "MinhaTask"
```

```
# Listar tarefas em pasta específica
Get-ScheduledTask -TaskPath "\Microsoft\Windows\WindowsUpdate\"

# Detalhes de uma tarefa
$task = Get-ScheduledTask -TaskName "MinhaTask"
$task | Format-List *

# Informações de última execução
Get-ScheduledTaskInfo -TaskName "MinhaTask"

# Filtrar tarefas
Get-ScheduledTask | Where-Object {
    $_.TaskName -like "*Backup*" -and
    $_.State -eq 'Ready'
} | Select-Object TaskName, State, TaskPath
```

4.2.3 Criando Tarefas Agendadas

Componentes de uma Tarefa

```
# 1. ACTION - O que executar
$action = New-ScheduledTaskAction

# 2. TRIGGER - Quando executar
$trigger = New-ScheduledTaskTrigger

# 3. PRINCIPAL - Com quais credenciais
$principal = New-ScheduledTaskPrincipal

# 4. SETTINGS - Configurações adicionais
```

```
$settings = New-ScheduledTaskSettingsSet

# 5. REGISTRAR a tarefa
Register-ScheduledTask
```

Exemplo 1: Tarefa Simples - Executar Script Diariamente

```
# Definir ação: executar script PowerShell
$action = New-ScheduledTaskAction `
  -Execute "PowerShell.exe" `
  -Argument "-NoProfile -ExecutionPolicy Bypass -File C:\Scripts\Backup.ps1"

# Definir trigger: diariamente às 22:00
$trigger = New-ScheduledTaskTrigger -Daily -At "22:00"

# Definir configurações
$settings = New-ScheduledTaskSettingsSet `
  -AllowStartIfOnBatteries `
  -DontStopIfGoingOnBatteries `
  -StartWhenAvailable

# Registrar tarefa
Register-ScheduledTask `
  -TaskName "Backup Diário" `
  -Description "Executa backup diário dos dados" `
  -Action $action `
  -Trigger $trigger `
  -Settings $settings `
  -User "SYSTEM"
```

```
Write-Host "Tarefa criada com sucesso!"
```

Exemplo 2: Múltiplos Triggers

```
# Ação
$action = New-ScheduledTaskAction `
  -Execute "PowerShell.exe" `
  -Argument "-File C:\Scripts\Monitor.ps1"

# Múltiplos triggers
$trigger1 = New-ScheduledTaskTrigger -AtStartup
$trigger2 = New-ScheduledTaskTrigger -Daily -At "08:00"
$trigger3 = New-ScheduledTaskTrigger -Daily -At "18:00"

# Registrar com múltiplos triggers
Register-ScheduledTask `
  -TaskName "Monitoramento Sistema" `
  -Action $action `
  -Trigger $trigger1, $trigger2, $trigger3 `
  -User "SYSTEM"
```

Exemplo 3: Trigger por Evento

```
# Criar trigger baseado em evento do log
$trigger = New-ScheduledTaskTrigger `
  -AtLogOn

# Ou trigger por evento específico
$eventFilter = @"
<QueryList>
 <Query Id="0" Path="System">
  <Select Path="System">
```

```
    *[System[Provider[@Name='Microsoft-Windows-Power-Troubleshooter'] and (EventID=1)]]
  </Select>
 </Query>
</QueryList>
"@

# Criar CIM instance para trigger de evento
$class = Get-CimClass -ClassName MSFT_TaskEventTrigger -Namespace Root/Microsoft/Windows/TaskScheduler
$trigger = New-CimInstance -CimClass $class -ClientOnly
$trigger.Subscription = $eventFilter
$trigger.Enabled = $true

# Ação
$action = New-ScheduledTaskAction `
  -Execute "PowerShell.exe" `
  -Argument "-File C:\Scripts\TratarEvento.ps1"

# Registrar
Register-ScheduledTask `
  -TaskName "Responder Evento Sistema" `
  -Action $action `
  -Trigger $trigger
```

4.2.4 Tipos de Triggers

```
# Trigger único (uma vez)
$trigger = New-ScheduledTaskTrigger -Once -At "2025-12-31 23:59"

# Trigger diário
```

```
$trigger = New-ScheduledTaskTrigger -Daily -At "03:00"

# Trigger semanal (múltiplos dias)
$trigger = New-ScheduledTaskTrigger -Weekly -DaysOfWeek Monday, Wednesday, Friday -At "18:00"

# Trigger na inicialização
$trigger = New-ScheduledTaskTrigger -AtStartup

# Trigger no logon
$trigger = New-ScheduledTaskTrigger -AtLogOn

# Trigger com repetição
$trigger = New-ScheduledTaskTrigger -Once -At "08:00" -RepetitionInterval (New-TimeSpan -Hours 1) -RepetitionDuration (New-TimeSpan -Hours 12)

# Trigger com atraso
$trigger = New-ScheduledTaskTrigger -Daily -At "00:00"
$trigger.Delay = "PT15M"  # Atraso de 15 minutos
```

4.2.5 Configurações Avançadas

```
# Settings completos
$settings = New-ScheduledTaskSettingsSet `
  -AllowStartIfOnBatteries `            # Permitir iniciar em bateria
  -DontStopIfGoingOnBatteries `         # Não parar se entrar em bateria
  -StartWhenAvailable `                 # Iniciar quando possível se perdeu horário
  -RunOnlyIfNetworkAvailable `          # Executar apenas se houver rede
  -ExecutionTimeLimit (New-TimeSpan -Hours 2) `  # Tempo máximo de execução
  -RestartCount 3 `                     # Tentativas de restart em falha
  -RestartInterval (New-TimeSpan -Minutes 5) `   # Intervalo entre tentativas
```

```
    -MultipleInstances IgnoreNew          # Comportamento se já estiver executando

# Outras opções de MultipleInstances
# - Parallel: Executar em paralelo
# - Queue: Enfileirar
# - StopExisting: Parar existente e iniciar nova
# - IgnoreNew: Ignorar nova instância (padrão)
```

4.2.6 Credenciais e Permissões

```
# Executar como SYSTEM
Register-ScheduledTask `
    -TaskName "MinhaTask" `
    -Action $action `
    -Trigger $trigger `
    -User "SYSTEM"

# Executar como usuário específico (solicita senha)
Register-ScheduledTask `
    -TaskName "MinhaTask" `
    -Action $action `
    -Trigger $trigger `
    -User "DOMINIO\Usuario"

# Executar com privilégios elevados
$principal = New-ScheduledTaskPrincipal `
    -UserId "SYSTEM" `
    -LogonType ServiceAccount `
    -RunLevel Highest
```

```
Register-ScheduledTask `
  -TaskName "MinhaTask" `
  -Action $action `
  -Trigger $trigger `
  -Principal $principal

# Executar independente do usuário estar logado
$principal = New-ScheduledTaskPrincipal `
  -UserId "DOMINIO\Usuario" `
  -LogonType Password `
  -RunLevel Limited

# Usar credencial armazenada
$cred = Get-Credential
Register-ScheduledTask `
  -TaskName "MinhaTask" `
  -Action $action `
  -Trigger $trigger `
  -User $cred.UserName `
  -Password $cred.GetNetworkCredential().Password
```

4.2.7 Modificar e Gerenciar Tarefas Existentes

```
# Obter tarefa existente
$task = Get-ScheduledTask -TaskName "MinhaTask"

# Modificar trigger
$newTrigger = New-ScheduledTaskTrigger -Daily -At "05:00"
Set-ScheduledTask -TaskName "MinhaTask" -Trigger $newTrigger
```

```
# Modificar ação
$newAction = New-ScheduledTaskAction `
  -Execute "PowerShell.exe" `
  -Argument "-File C:\Scripts\NovoScript.ps1"
Set-ScheduledTask -TaskName "MinhaTask" -Action $newAction

# Desabilitar tarefa
Disable-ScheduledTask -TaskName "MinhaTask"

# Habilitar tarefa
Enable-ScheduledTask -TaskName "MinhaTask"

# Executar tarefa manualmente
Start-ScheduledTask -TaskName "MinhaTask"

# Parar execução
Stop-ScheduledTask -TaskName "MinhaTask"

# Remover tarefa
Unregister-ScheduledTask -TaskName "MinhaTask" -Confirm:$false

# Exportar tarefa para XML
$task = Get-ScheduledTask -TaskName "MinhaTask"
$task | Export-ScheduledTask | Out-File "C:\Backup\MinhaTask.xml"

# Importar tarefa de XML
Register-ScheduledTask -Xml (Get-Content "C:\Backup\MinhaTask.xml" | Out-String) -TaskName "MinhaTask"
```

4.2.8 Exemplo Completo: Sistema de Tarefas Agendadas

```
# Arquivo: GerenciadorTarefas.ps1

<#
.SYNOPSIS
  Sistema completo de gerenciamento de tarefas agendadas
#>

# Função para criar tarefa de backup
function New-BackupTask {
  param(
    [string]$BackupPath = "C:\Backups",
    [string]$SourcePath = "C:\Dados",
    [string]$Horario = "22:00"
  )

  Write-Host "Criando tarefa de backup..." -ForegroundColor Yellow

  # Script de backup inline
  $scriptBackup = @"
`$origem = '$SourcePath'
`$destino = '$BackupPath'
`$timestamp = Get-Date -Format 'yyyyMMdd_HHmmss'
`$nomeBackup = "Backup_`$timestamp.zip"
`$caminhoCompleto = Join-Path `$destino `$nomeBackup

if (-not (Test-Path `$destino)) {
  New-Item -Path `$destino -ItemType Directory -Force | Out-Null
```

```
}

try {
    Compress-Archive -Path `$origem -DestinationPath `$caminhoCompleto -CompressionLevel Optimal
    Write-Output "Backup criado: `$caminhoCompleto"

    # Limpar backups antigos (manter últimos 7)
    Get-ChildItem -Path `$destino -Filter "Backup_*.zip" |
        Sort-Object CreationTime -Descending |
        Select-Object -Skip 7 |
        Remove-Item -Force
}
catch {
    Write-Error "Erro no backup: `$_"
    exit 1
}
"@

    # Salvar script
    $scriptPath = "C:\Scripts\BackupAutomatico.ps1"
    if (-not (Test-Path "C:\Scripts")) {
        New-Item -Path "C:\Scripts" -ItemType Directory -Force | Out-Null
    }
    $scriptBackup | Out-File -FilePath $scriptPath -Encoding UTF8 -Force

    # Criar tarefa
    $action = New-ScheduledTaskAction `
```

```
        -Execute "PowerShell.exe" `
        -Argument "-NoProfile -ExecutionPolicy Bypass -File `"$scriptPath`""

    $trigger = New-ScheduledTaskTrigger -Daily -At $Horario

    $settings = New-ScheduledTaskSettingsSet `
        -AllowStartIfOnBatteries `
        -DontStopIfGoingOnBatteries `
        -StartWhenAvailable `
        -ExecutionTimeLimit (New-TimeSpan -Hours 2)

    Register-ScheduledTask `
        -TaskName "Backup Automático Dados" `
        -Description "Backup diário automático em $Horario" `
        -Action $action `
        -Trigger $trigger `
        -Settings $settings `
        -User "SYSTEM" `
        -Force

    Write-Host "✓ Tarefa de backup criada com sucesso!" -ForegroundColor Green
    Write-Host "  Horário: $Horario" -ForegroundColor Cyan
    Write-Host "  Origem: $SourcePath" -ForegroundColor Cyan
    Write-Host "  Destino: $BackupPath" -ForegroundColor Cyan
}

# Função para criar tarefa de limpeza
function New-CleanupTask {
```

```
    param(
        [string]$TempPath = "C:\Temp",
        [int]$DaysOld = 30
    )

    Write-Host "Criando tarefa de limpeza..." -ForegroundColor Yellow

    $scriptLimpeza = @"
`$caminho = '$TempPath'
`$dias = $DaysOld
`$dataLimite = (Get-Date).AddDays(-`$dias)

if (Test-Path `$caminho) {
    `$arquivos = Get-ChildItem -Path `$caminho -Recurse -File |
                Where-Object { `$_.LastWriteTime -lt `$dataLimite }

    `$totalArquivos = `$arquivos.Count
    `$tamanhoTotal = (`$arquivos | Measure-Object -Property Length -Sum).Sum / 1MB

    Write-Output "Encontrados `$totalArquivos arquivos antigos"
    Write-Output "Espaço a liberar: `$([math]::Round(`$tamanhoTotal, 2)) MB"

    `$arquivos | Remove-Item -Force -ErrorAction SilentlyContinue

    Write-Output "Limpeza concluída!"
}
"@
```

```
    $scriptPath = "C:\Scripts\LimpezaAutomatica.ps1"
    $scriptLimpeza | Out-File -FilePath $scriptPath -Encoding UTF8 -Force

    $action = New-ScheduledTaskAction `
        -Execute "PowerShell.exe" `
        -Argument "-NoProfile -ExecutionPolicy Bypass -File `"$scriptPath`""

    # Executar semanalmente aos domingos à 01:00
    $trigger = New-ScheduledTaskTrigger -Weekly -DaysOfWeek Sunday -At "01:00"

    $settings = New-ScheduledTaskSettingsSet `
        -AllowStartIfOnBatteries `
        -DontStopIfGoingOnBatteries `
        -StartWhenAvailable

    Register-ScheduledTask `
        -TaskName "Limpeza Arquivos Temporários" `
        -Description "Remove arquivos temporários com mais de $DaysOld dias" `
        -Action $action `
        -Trigger $trigger `
        -Settings $settings `
        -User "SYSTEM" `
        -Force

    Write-Host "✓ Tarefa de limpeza criada com sucesso!" -ForegroundColor Green
}
```

```
# Função para criar tarefa de monitoramento
function New-MonitoringTask {
    param(
        [int]$IntervalMinutes = 15
    )

    Write-Host "Criando tarefa de monitoramento..." -ForegroundColor Yellow

    $scriptMonitor = @"
`$alertas = @()

# Verificar CPU
`$cpu = (Get-CimInstance Win32_Processor).LoadPercentage
if (`$cpu -gt 90) {
    `$alertas += "CPU alta: `$cpu%"
}

# Verificar Memória
`$os = Get-CimInstance Win32_OperatingSystem
`$memoriaUsada = [math]::Round(((`$os.TotalVisibleMemorySize -
`$os.FreePhysicalMemory) / `$os.TotalVisibleMemorySize) * 100, 2)
if (`$memoriaUsada -gt 90) {
    `$alertas += "Memória alta: `$memoriaUsada%"
}

# Verificar Disco
`$discos = Get-CimInstance Win32_LogicalDisk -Filter "DriveType=3"
foreach (`$disco in `$discos) {
```

```
    `$percentUsado = [math]::Round(((`$disco.Size - `$disco.FreeSpace) / `$disco.Size) * 100, 2)
    if (`$percentUsado -gt 90) {
        `$alertas += "Disco `$(`$disco.DeviceID) cheio: `$percentUsado%"
    }
}

if (`$alertas.Count -gt 0) {
    `$logPath = "C:\Logs\Alertas_`$(Get-Date -Format 'yyyyMMdd').log"
    if (-not (Test-Path (Split-Path `$logPath))) {
        New-Item -Path (Split-Path `$logPath) -ItemType Directory -Force | Out-Null
    }

    `$timestamp = Get-Date -Format 'yyyy-MM-dd HH:mm:ss'
    foreach (`$alerta in `$alertas) {
        "[`$timestamp] ALERTA: `$alerta" | Out-File `$logPath -Append
    }
}
"@

    $scriptPath = "C:\Scripts\MonitoramentoSistema.ps1"
    $scriptMonitor | Out-File -FilePath $scriptPath -Encoding UTF8 -Force

    $action = New-ScheduledTaskAction `
        -Execute "PowerShell.exe" `
        -Argument "-NoProfile -ExecutionPolicy Bypass -File `"$scriptPath`""

    # Trigger repetido a cada X minutos
```

```
    $trigger = New-ScheduledTaskTrigger `
        -Once -At "00:00" `
        -RepetitionInterval (New-TimeSpan -Minutes $IntervalMinutes) `
        -RepetitionDuration ([TimeSpan]::MaxValue)

    $settings = New-ScheduledTaskSettingsSet `
        -AllowStartIfOnBatteries `
        -DontStopIfGoingOnBatteries `
        -ExecutionTimeLimit (New-TimeSpan -Minutes 5) `
        -MultipleInstances IgnoreNew

    Register-ScheduledTask `
        -TaskName "Monitoramento Sistema" `
        -Description "Monitora recursos do sistema a cada $IntervalMinutes minutos" `
        -Action $action `
        -Trigger $trigger `
        -Settings $settings `
        -User "SYSTEM" `
        -Force

    Write-Host "✓ Tarefa de monitoramento criada com sucesso!" -ForegroundColor Green
    Write-Host "  Intervalo: $IntervalMinutes minutos" -ForegroundColor Cyan
}

# Função para listar todas as tarefas customizadas
function Get-CustomTasks {
    $taskNames = @(
```

```
        "Backup Automático Dados",
        "Limpeza Arquivos Temporários",
        "Monitoramento Sistema"
    )

    Write-Host "`n=== TAREFAS CUSTOMIZADAS ===" -ForegroundColor Cyan

    foreach ($taskName in $taskNames) {
        $task = Get-ScheduledTask -TaskName $taskName -ErrorAction SilentlyContinue
        if ($task) {
            $info = Get-ScheduledTaskInfo -TaskName $taskName

            Write-Host "`n[$taskName]" -ForegroundColor Yellow
            Write-Host "  Estado: $($task.State)" -ForegroundColor $(if ($task.State -eq 'Ready') {'Green'} else {'Red'})
            Write-Host "  Última Execução: $($info.LastRunTime)"
            Write-Host "  Próxima Execução: $($info.NextRunTime)"
            Write-Host "  Último Resultado: $($info.LastTaskResult)"
        }
        else {
            Write-Host "`n[$taskName]" -ForegroundColor Yellow
            Write-Host "  Status: NÃO CRIADA" -ForegroundColor Red
        }
    }
}

# Função para remover todas as tarefas customizadas
function Remove-AllCustomTasks {
```

```
    $taskNames = @(
        "Backup Automático Dados",
        "Limpeza Arquivos Temporários",
        "Monitoramento Sistema"
    )

    Write-Host "Removendo tarefas customizadas..." -ForegroundColor Yellow

    foreach ($taskName in $taskNames) {
        try {
            Unregister-ScheduledTask -TaskName $taskName -Confirm:$false -
ErrorAction Stop
            Write-Host "✓ $taskName removida" -ForegroundColor Green
        }
        catch {
            Write-Host "✗ $taskName não encontrada" -ForegroundColor Gray
        }
    }
}

# Menu interativo
function Show-Menu {
    Clear-Host
    Write-Host
"╔═══════════════════════════════════════════════════════╗" -
ForegroundColor Cyan
    Write-Host "║   GERENCIADOR DE TAREFAS AGENDADAS    ║" -ForegroundColor
Cyan
```

```
    Write-Host
"╚═══════════════════════════════════════════════════════════════════════════╝" -
ForegroundColor Cyan
    Write-Host ""
    Write-Host "1. Criar Tarefa de Backup" -ForegroundColor White
    Write-Host "2. Criar Tarefa de Limpeza" -ForegroundColor White
    Write-Host "3. Criar Tarefa de Monitoramento" -ForegroundColor White
    Write-Host "4. Listar Tarefas Customizadas" -ForegroundColor White
    Write-Host "5. Remover Todas as Tarefas" -ForegroundColor White
    Write-Host "6. Sair" -ForegroundColor White
    Write-Host ""
}

# Loop do menu
do {
    Show-Menu
    $opcao = Read-Host "Escolha uma opção"

    switch ($opcao) {
        "1" {
            New-BackupTask
            Read-Host "`nPressione Enter para continuar"
        }
        "2" {
            New-CleanupTask
            Read-Host "`nPressione Enter para continuar"
        }
        "3" {
            New-MonitoringTask
```

```
            Read-Host "`nPressione Enter para continuar"
        }
        "4" {
            Get-CustomTasks
            Read-Host "`nPressione Enter para continuar"
        }
        "5" {
            $confirmacao = Read-Host "Tem certeza? (S/N)"
            if ($confirmacao -eq 'S') {
                Remove-AllCustomTasks
            }
            Read-Host "`nPressione Enter para continuar"
        }
        "6" {
            Write-Host "Encerrando..." -ForegroundColor Cyan
        }
        default {
            Write-Host "Opção inválida!" -ForegroundColor Red
            Start-Sleep -Seconds 2
        }
    }
} while ($opcao -ne "6")
```

Conclusão da Seção 4

Nesta seção, exploramos em profundidade:

1. **Fundamentos de Scripting**:
    - Estrutura básica de scripts
    - Parâmetros e validação
    - Funções avançadas

2. **Técnicas Avançadas**:
    - Tratamento robusto de erros
    - Sistema de logging profissional
    - Debugging e troubleshooting
3. **Tarefas Agendadas**:
    - Criação e gerenciamento de tarefas
    - Tipos de triggers
    - Configurações avançadas
    - Sistema completo de automação

Com essas habilidades, você pode criar soluções de automação robustas, confiáveis e profissionais que executam tarefas complexas de forma

automatizada e eficiente.

5. ADMINISTRAÇÃO DE SISTEMAS OPERACIONAIS WINDOWS

5.1 Gerenciamento de Arquivos, Processos e Serviços

5.1.1 Gerenciamento de Arquivos e Diretórios

Navegação e Listagem

```
# Get-Location (pwd): Obter diretório atual
Get-Location
pwd  # Alias

# Set-Location (cd): Mudar diretório
Set-Location C:\Windows
cd C:\Windows  # Alias
sl C:\Windows  # Alias

# Navegar para diretório anterior
Set-Location -Path $PSHOME
```

```
# Voltar ao diretório home
Set-Location ~

# Push-Location e Pop-Location: Pilha de diretórios
Push-Location C:\Windows
Push-Location C:\Temp
Get-Location  # C:\Temp
Pop-Location  # Volta para C:\Windows
Pop-Location  # Volta para o diretório inicial

# Get-ChildItem (ls, dir): Listar arquivos e diretórios
Get-ChildItem
ls  # Alias Unix
dir  # Alias DOS
gci  # Alias PowerShell

# Listar com detalhes
Get-ChildItem -Force  # Incluir ocultos
Get-ChildItem -Recurse  # Recursivo (subdiretórios)
Get-ChildItem -File  # Apenas arquivos
Get-ChildItem -Directory  # Apenas diretórios

# Filtrar por extensão
Get-ChildItem -Filter *.txt
Get-ChildItem -Include *.log, *.txt -Recurse
Get-ChildItem -Exclude *.tmp

# Listar com profundidade específica
```

```
Get-ChildItem -Recurse -Depth 2

# Listar arquivos ocultos e sistema
Get-ChildItem -Force -Attributes Hidden
Get-ChildItem -Attributes System

# Ordenar resultados
Get-ChildItem | Sort-Object Length -Descending
Get-ChildItem | Sort-Object LastWriteTime
```

Criação de Arquivos e Diretórios

```
# New-Item: Criar arquivos e diretórios

# Criar arquivo vazio
New-Item -Path "arquivo.txt" -ItemType File

# Criar arquivo com conteúdo
New-Item -Path "dados.txt" -ItemType File -Value "Conteúdo inicial"

# Criar diretório
New-Item -Path "NovaPasta" -ItemType Directory

# Criar estrutura de diretórios
New-Item -Path "Projeto\Src\Controllers" -ItemType Directory -Force

# Criar múltiplos arquivos
1..5 | ForEach-Object {
   New-Item -Path "arquivo$_.txt" -ItemType File
}
```

 # Criar arquivo temporário

$tempFile = New-TemporaryFile

Write-Host "Arquivo temporário: $($tempFile.FullName)"

 # Mkdir (alias para criar diretórios)

mkdir "OutraPasta"

md "MaisPasta"

Cópia, Movimentação e Renomeação

 # Copy-Item: Copiar arquivos e diretórios

 # Copiar arquivo

Copy-Item -Path "origem.txt" -Destination "destino.txt"

cp origem.txt destino.txt *# Alias*

 # Copiar para outro diretório

Copy-Item -Path "arquivo.txt" -Destination "C:\Backup\"

 # Copiar diretório recursivamente

Copy-Item -Path "Pasta" -Destination "PastaCopia" -Recurse

 # Copiar múltiplos arquivos

Copy-Item -Path "*.txt" -Destination "C:\Documentos\"

 # Copiar preservando estrutura

Copy-Item -Path "C:\Projeto*" -Destination "D:\Backup\Projeto\" -Recurse -Force

 # Copiar com filtro

```
Get-ChildItem -Filter "*.log" | Copy-Item -Destination "C:\Logs\"

# Move-Item: Mover/renomear arquivos

# Mover arquivo
Move-Item -Path "arquivo.txt" -Destination "C:\Destino\"
mv arquivo.txt C:\Destino\  # Alias

# Renomear arquivo
Move-Item -Path "antigo.txt" -Destination "novo.txt"

# Mover diretório
Move-Item -Path "Pasta" -Destination "C:\NovoLocal\"

# Mover com sobrescrita
Move-Item -Path "arquivo.txt" -Destination "C:\Destino\" -Force

# Rename-Item: Renomear especificamente

# Renomear arquivo
Rename-Item -Path "velho.txt" -NewName "novo.txt"
ren velho.txt novo.txt  # Alias

# Renomear diretório
Rename-Item -Path "PastaAntiga" -NewName "PastaNova"

# Renomear múltiplos arquivos (adicionar prefixo)
Get-ChildItem -Filter "*.txt" | Rename-Item -NewName { "Backup_" + $_.Name }
```

 # Renomear múltiplos arquivos (trocar extensão)

Get-ChildItem -Filter "*.txt" | Rename-Item -NewName { $_.Name -replace '\.txt$', '.log' }

Exclusão de Arquivos e Diretórios

 # Remove-Item: Remover arquivos e diretórios

 # Remover arquivo

Remove-Item -Path "arquivo.txt"

rm arquivo.txt *# Alias Unix*

del arquivo.txt *# Alias DOS*

 # Remover diretório vazio

Remove-Item -Path "Pasta"

 # Remover diretório com conteúdo

Remove-Item -Path "Pasta" -Recurse

 # Remover com confirmação

Remove-Item -Path "arquivo.txt" -Confirm

 # Remover forçadamente (incluindo readonly)

Remove-Item -Path "arquivo.txt" -Force

 # Remover múltiplos arquivos

Remove-Item -Path "*.tmp"

Get-ChildItem -Filter "*.log" | Remove-Item

```
# Simular remoção (WhatIf)
Remove-Item -Path "Pasta" -Recurse -WhatIf

# Remover arquivos antigos
Get-ChildItem -Filter "*.log" |
  Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-30) } |
  Remove-Item

# Clear-RecycleBin: Esvaziar lixeira
Clear-RecycleBin -DriveLetter C -Force
Clear-RecycleBin -Force  # Todas as unidades
```

Leitura e Escrita de Arquivos

```
# Get-Content: Ler conteúdo de arquivo

# Ler arquivo completo
Get-Content -Path "arquivo.txt"
cat arquivo.txt  # Alias Unix
type arquivo.txt  # Alias DOS

# Ler com encoding específico
Get-Content -Path "arquivo.txt" -Encoding UTF8

# Ler últimas N linhas
Get-Content -Path "log.txt" -Tail 10

# Ler primeiras N linhas
Get-Content -Path "dados.txt" -TotalCount 5
```

```
# Ler como stream (linha por linha)
Get-Content -Path "grande.txt" -ReadCount 100

# Monitorar arquivo em tempo real (tail -f)
Get-Content -Path "log.txt" -Wait -Tail 10

# Ler como bytes
Get-Content -Path "imagem.jpg" -Encoding Byte

# Set-Content: Escrever (sobrescrever) arquivo

# Escrever texto
Set-Content -Path "arquivo.txt" -Value "Novo conteúdo"
"Texto direto" | Set-Content "arquivo.txt"

# Escrever múltiplas linhas
Set-Content -Path "lista.txt" -Value @("Linha 1", "Linha 2", "Linha 3")

# Escrever com encoding
Set-Content -Path "arquivo.txt" -Value "Conteúdo" -Encoding UTF8

# Add-Content: Adicionar ao arquivo

# Adicionar texto ao final
Add-Content -Path "log.txt" -Value "Nova entrada de log"
"Mais uma linha" | Add-Content "arquivo.txt"

# Adicionar timestamp
```

Add-Content -Path "log.txt" -Value "$(Get-Date) - Evento registrado"

 # Out-File: Redirecionar saída para arquivo

 # Escrever saída de comando

Get-Process | Out-File "processos.txt"

 # Append

Get-Service | Out-File "servicos.txt" -Append

 # Com encoding

Get-ChildItem | Out-File "lista.txt" -Encoding UTF8

 # Com largura específica

Get-Process | Out-File "processos.txt" -Width 200

Manipulação de Conteúdo

 # Select-String: Buscar texto em arquivos (grep)

 # Buscar padrão em arquivo

Select-String -Path "log.txt" -Pattern "erro"

 # Buscar em múltiplos arquivos

Select-String -Path "*.txt" -Pattern "senha"

 # Case-sensitive

Select-String -Path "dados.txt" -Pattern "PowerShell" -CaseSensitive

 # Buscar com regex

```
Select-String -Path "log.txt" -Pattern "\d{3}\.\d{3}\.\d{3}-\d{2}"

# Contexto (linhas antes/depois)
Select-String -Path "log.txt" -Pattern "erro" -Context 2,3

# Buscar recursivamente
Get-ChildItem -Recurse -Filter "*.log" | Select-String -Pattern "falha"

# Exibir apenas arquivos que contêm o padrão
Select-String -Path "*.txt" -Pattern "importante" -List

# Compare-Object: Comparar arquivos

# Comparar conteúdo de dois arquivos
$arquivo1 = Get-Content "versao1.txt"
$arquivo2 = Get-Content "versao2.txt"
Compare-Object -ReferenceObject $arquivo1 -DifferenceObject $arquivo2

# Mostrar apenas diferenças
Compare-Object $arquivo1 $arquivo2 -PassThru

# Incluir linhas iguais
Compare-Object $arquivo1 $arquivo2 -IncludeEqual

# Measure-Object: Estatísticas de arquivo

# Contar linhas em arquivo
Get-Content "arquivo.txt" | Measure-Object -Line
```

 # Contar palavras e caracteres

Get-Content "documento.txt" | Measure-Object -Word -Character -Line

Atributos e Propriedades de Arquivos

 # Get-Item e Get-ItemProperty: Obter informações

 # Obter informações de arquivo

$arquivo = Get-Item "documento.txt"

$arquivo.Name

$arquivo.Length

$arquivo.CreationTime

$arquivo.LastWriteTime

$arquivo.LastAccessTime

$arquivo.Extension

$arquivo.FullName

$arquivo.Directory

$arquivo.Attributes

 # Obter propriedades específicas

Get-ItemProperty -Path "arquivo.txt" -Name LastWriteTime

 # Set-ItemProperty: Modificar propriedades

 # Tornar arquivo readonly

Set-ItemProperty -Path "importante.txt" -Name IsReadOnly -Value $true

 # Remover readonly

Set-ItemProperty -Path "importante.txt" -Name IsReadOnly -Value $false

```
# Modificar atributos
Set-ItemProperty -Path "arquivo.txt" -Name Attributes -Value "Hidden"
Set-ItemProperty -Path "arquivo.txt" -Name Attributes -Value "Archive"

# Modificar timestamps
$arquivo = Get-Item "teste.txt"
$arquivo.CreationTime = "2025-01-01"
$arquivo.LastWriteTime = Get-Date

# Test-Path: Verificar existência

# Verificar se arquivo existe
Test-Path "arquivo.txt"

# Verificar se diretório existe
Test-Path "C:\Pasta" -PathType Container

# Verificar se é arquivo
Test-Path "documento.txt" -PathType Leaf

# Resolve-Path: Resolver caminho completo
Resolve-Path ".\arquivo.txt"

# Split-Path: Manipular caminhos
Split-Path "C:\Pasta\arquivo.txt" -Parent  # C:\Pasta
Split-Path "C:\Pasta\arquivo.txt" -Leaf    # arquivo.txt
Split-Path "C:\Pasta\arquivo.txt" -Extension  # .txt
```

 # Join-Path: Combinar caminhos

Join-Path "C:\Dados" "arquivo.txt" *# C:\Dados\arquivo.txt*

Compressão e Descompressão

 # Compress-Archive: Criar arquivo ZIP

 # Comprimir arquivo

Compress-Archive -Path "arquivo.txt" -DestinationPath "arquivo.zip"

 # Comprimir diretório

Compress-Archive -Path "Pasta" -DestinationPath "backup.zip"

 # Comprimir múltiplos itens

Compress-Archive -Path "*.txt", "*.log" -DestinationPath "arquivos.zip"

 # Adicionar a arquivo existente

Compress-Archive -Path "novo.txt" -DestinationPath "arquivo.zip" -Update

 # Nível de compressão

Compress-Archive -Path "Dados" -DestinationPath "dados.zip" -CompressionLevel Optimal

 # Níveis: Fastest, Optimal, NoCompression

 # Expand-Archive: Extrair arquivo ZIP

 # Extrair arquivo

Expand-Archive -Path "arquivo.zip" -DestinationPath "C:\Extraidos"

```
# Extrair sobrescrevendo
Expand-Archive -Path "backup.zip" -DestinationPath "C:\Restore" -Force

# Listar conteúdo do ZIP (sem extrair)
Add-Type -AssemblyName System.IO.Compression.FileSystem
$zip = [System.IO.Compression.ZipFile]::OpenRead("arquivo.zip")
$zip.Entries | Select-Object Name, Length, CompressedLength
$zip.Dispose()
```

Exemplo Completo: Sistema de Gerenciamento de Arquivos

```
# Arquivo: GerenciadorArquivos.ps1

<#
.SYNOPSIS
   Sistema completo de gerenciamento de arquivos
#>

# Função para organizar arquivos por extensão
function Organize-FilesByExtension {
   param(
      [Parameter(Mandatory=$true)]
      [ValidateScript({Test-Path $_})]
      [string]$Path,

      [switch]$WhatIf
   )

   Write-Host "`nOrganizando arquivos em: $Path" -ForegroundColor Cyan
```

```
  $arquivos = Get-ChildItem -Path $Path -File

  if ($arquivos.Count -eq 0) {
    Write-Host "Nenhum arquivo encontrado." -ForegroundColor Yellow
    return
  }

  $agrupados = $arquivos | Group-Object Extension

  foreach ($grupo in $agrupados) {
    $extensao = if ($grupo.Name) { $grupo.Name.TrimStart('.') } else {
"SemExtensao" }
    $pastaDestino = Join-Path $Path $extensao

    if (-not (Test-Path $pastaDestino)) {
      Write-Host "Criando pasta: $extensao" -ForegroundColor Yellow
      if (-not $WhatIf) {
        New-Item -Path $pastaDestino -ItemType Directory | Out-Null
      }
    }

    Write-Host "` nMovendo $($grupo.Count) arquivo(s) .$extensao" -
ForegroundColor Green

    foreach ($arquivo in $grupo.Group) {
      $destino = Join-Path $pastaDestino $arquivo.Name
      Write-Host "  $($arquivo.Name) → $extensao\" -ForegroundColor Gray

      if (-not $WhatIf) {
```

```
            Move-Item -Path $arquivo.FullName -Destination $destino -Force
        }
    }
}

Write-Host "`nOrganização concluída!" -ForegroundColor Green
}

# Função para encontrar arquivos duplicados
function Find-DuplicateFiles {
    param(
        [Parameter(Mandatory=$true)]
        [ValidateScript({Test-Path $_})]
        [string]$Path,

        [switch]$Recurse
    )

    Write-Host "`nBuscando arquivos duplicados..." -ForegroundColor Cyan

    $parametros = @{
        Path = $Path
        File = $true
    }
    if ($Recurse) { $parametros.Recurse = $true }

    $arquivos = Get-ChildItem @parametros
```

```
  Write-Host "Calculando hashes de $($arquivos.Count) arquivos..." -
ForegroundColor Yellow

  $hashes = $arquivos | ForEach-Object {
    [PSCustomObject]@{
      Path = $_.FullName
      Name = $_.Name
      Size = $_.Length
      Hash = (Get-FileHash -Path $_.FullName -Algorithm MD5).Hash
    }
  }

  $duplicados = $hashes | Group-Object Hash | Where-Object Count -gt 1

  if ($duplicados.Count -eq 0) {
    Write-Host "Nenhum arquivo duplicado encontrado." -ForegroundColor Green
    return
  }

  Write-Host "`nEncontrados $($duplicados.Count) conjuntos de duplicados:" -
ForegroundColor Red

  $espacoDesperdicio = 0

  foreach ($grupo in $duplicados) {
    $tamanho = $grupo.Group[0].Size
    $espacoDesperdicio += $tamanho * ($grupo.Count - 1)
```

```
    Write-Host "`n[$($grupo.Group[0].Name)] - $($grupo.Count) cópias - $([math]::Round($tamanho/1MB, 2)) MB" -ForegroundColor Yellow
    foreach ($item in $grupo.Group) {
      Write-Host "  $($item.Path)" -ForegroundColor Gray
    }
  }

  Write-Host "`nEspaço desperdiçado: $([math]::Round($espacoDesperdicio/1MB, 2)) MB" -ForegroundColor Red

  return $duplicados
}

# Função para limpar arquivos temporários
function Clear-TemporaryFiles {
  param(
    [int]$DaysOld = 7,
    [switch]$WhatIf
  )

  $caminhos = @(
    $env:TEMP,
    "C:\Windows\Temp",
    "C:\Windows\Prefetch"
  )

  $dataLimite = (Get-Date).AddDays(-$DaysOld)
  $totalRemovido = 0
  $espacoLiberado = 0
```

```
  Write-Host "`nLimpando arquivos temporários com mais de $DaysOld dias..." -ForegroundColor Cyan

  foreach ($caminho in $caminhos) {
    if (-not (Test-Path $caminho)) {
      continue
    }

    Write-Host "`nVerificando: $caminho" -ForegroundColor Yellow

    try {
      $arquivos = Get-ChildItem -Path $caminho -File -Recurse -ErrorAction SilentlyContinue |
                  Where-Object { $_.LastWriteTime -lt $dataLimite }

      foreach ($arquivo in $arquivos) {
        try {
          $tamanho = $arquivo.Length
          Write-Host "  Removendo: $($arquivo.Name)" -ForegroundColor Gray

          if (-not $WhatIf) {
            Remove-Item -Path $arquivo.FullName -Force -ErrorAction Stop
            $totalRemovido++
            $espacoLiberado += $tamanho
          }
        }
        catch {
          Write-Host "  Erro ao remover: $($arquivo.Name)" -ForegroundColor Red
```

```
            }
        }
    }
    catch {
        Write-Host "Erro ao acessar: $caminho" -ForegroundColor Red
    }
  }

  Write-Host "`nLimpeza concluída!" -ForegroundColor Green
  Write-Host "Arquivos removidos: $totalRemovido" -ForegroundColor Cyan
  Write-Host "Espaço liberado: $([math]::Round($espacoLiberado/1MB, 2)) MB" -ForegroundColor Cyan
}

# Função para criar relatório de uso de disco
function Get-DiskUsageReport {
  param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$Path,

    [int]$TopN = 20
  )

  Write-Host "`nAnalisando uso de disco em: $Path" -ForegroundColor Cyan

  $diretorios = Get-ChildItem -Path $Path -Directory -ErrorAction SilentlyContinue
  $relatorio = @()
```

```
  foreach ($dir in $diretorios) {
    Write-Host "Calculando: $($dir.Name)..." -ForegroundColor Gray

    try {
      $tamanho = (Get-ChildItem -Path $dir.FullName -Recurse -File -ErrorAction
SilentlyContinue |
            Measure-Object -Property Length -Sum).Sum

      $quantidade = (Get-ChildItem -Path $dir.FullName -Recurse -File -
ErrorAction SilentlyContinue |
              Measure-Object).Count

      $relatorio += [PSCustomObject]@{
        Diretorio = $dir.Name
        TamanhoMB = [math]::Round($tamanho / 1MB, 2)
        TamanhoGB = [math]::Round($tamanho / 1GB, 2)
        Arquivos = $quantidade
      }
    }
    catch {
      Write-Host "Erro ao processar: $($dir.Name)" -ForegroundColor Red
    }
  }

  $relatorio = $relatorio | Sort-Object TamanhoMB -Descending | Select-Object -
First $TopN

  Write-Host "`nTop $TopN diretórios por tamanho:" -ForegroundColor Green
```

```
    $relatorio | Format-Table -AutoSize

    $totalGB = ($relatorio | Measure-Object -Property TamanhoGB -Sum).Sum
    Write-Host "Total: $([math]::Round($totalGB, 2)) GB" -ForegroundColor Cyan

    return $relatorio
}

# Função para fazer backup inteligente
function New-SmartBackup {
    param(
        [Parameter(Mandatory=$true)]
        [ValidateScript({Test-Path $_})]
        [string]$Source,

        [Parameter(Mandatory=$true)]
        [string]$Destination,

        [int]$KeepBackups = 7,

        [string[]]$ExcludeExtensions = @("*.tmp", "*.temp", "*.cache")
    )

    Write-Host "`n=== BACKUP INTELIGENTE ===" -ForegroundColor Cyan
    Write-Host "Origem: $Source" -ForegroundColor Yellow
    Write-Host "Destino: $Destination" -ForegroundColor Yellow

    # Criar diretório de destino
```

```
    if (-not (Test-Path $Destination)) {
        New-Item -Path $Destination -ItemType Directory -Force | Out-Null
    }

    # Nome do backup
    $timestamp = Get-Date -Format "yyyyMMdd_HHmmss"
    $backupName = "Backup_$timestamp"
    $backupPath = Join-Path $Destination $backupName

    Write-Host "`nColetando arquivos..." -ForegroundColor Yellow

    # Obter arquivos (excluir extensões especificadas)
    $arquivos = Get-ChildItem -Path $Source -Recurse -File

    foreach ($extensao in $ExcludeExtensions) {
        $arquivos = $arquivos | Where-Object { $_.Name -notlike $extensao }
    }

    Write-Host "Arquivos a fazer backup: $($arquivos.Count)" -ForegroundColor
Cyan

    $tamanhoTotal = ($arquivos | Measure-Object -Property Length -Sum).Sum
    Write-Host "Tamanho total: $([math]::Round($tamanhoTotal/1MB, 2)) MB" -
ForegroundColor Cyan

    # Criar backup compactado
    Write-Host "`nCriando arquivo de backup..." -ForegroundColor Yellow

    try {
```

```
    Compress-Archive -Path $Source -DestinationPath "$backupPath.zip" -
CompressionLevel Optimal

    $backupFile = Get-Item "$backupPath.zip"
    $compressao = [math]::Round(($backupFile.Length / $tamanhoTotal) * 100, 2)

    Write-Host "Backup criado com sucesso!" -ForegroundColor Green
    Write-Host "Arquivo: $backupName.zip" -ForegroundColor Cyan
    Write-Host "Tamanho compactado:
$([math]::Round($backupFile.Length/1MB, 2)) MB" -ForegroundColor Cyan
    Write-Host "Taxa de compressão: $compressao%" -ForegroundColor Cyan

    # Limpar backups antigos
    Write-Host "`nLimpando backups antigos..." -ForegroundColor Yellow

    $backupsAntigos = Get-ChildItem -Path $Destination -Filter "Backup_*.zip" |
            Sort-Object CreationTime -Descending |
            Select-Object -Skip $KeepBackups

    foreach ($backup in $backupsAntigos) {
      Remove-Item -Path $backup.FullName -Force
      Write-Host "Removido: $($backup.Name)" -ForegroundColor Gray
    }

    Write-Host "`nBackup concluído com sucesso!" -ForegroundColor Green
  }
  catch {
    Write-Host "Erro ao criar backup: $_" -ForegroundColor Red
    return $false
```

```
    }

    return $true
}

# Menu principal
function Show-FileManagementMenu {
    do {
        Clear-Host
        Write-Host "╔═══════════════════════════════════════════════════════╗" -ForegroundColor Cyan
        Write-Host "║   SISTEMA DE GERENCIAMENTO ARQUIVOS   ║" -ForegroundColor Cyan
        Write-Host "╚═══════════════════════════════════════════════════════╝" -ForegroundColor Cyan
        Write-Host ""
        Write-Host "1. Organizar arquivos por extensão" -ForegroundColor White
        Write-Host "2. Encontrar arquivos duplicados" -ForegroundColor White
        Write-Host "3. Limpar arquivos temporários" -ForegroundColor White
        Write-Host "4. Relatório de uso de disco" -ForegroundColor White
        Write-Host "5. Criar backup inteligente" -ForegroundColor White
        Write-Host "6. Sair" -ForegroundColor White
        Write-Host ""

        $opcao = Read-Host "Escolha uma opção"

        switch ($opcao) {
            "1" {
```

```
            $caminho = Read-Host "Digite o caminho do diretório"
            if (Test-Path $caminho) {
                Organize-FilesByExtension -Path $caminho
            } else {
                Write-Host "Caminho não existe!" -ForegroundColor Red
            }
            Read-Host "`nPressione Enter para continuar"
        }
        "2" {
            $caminho = Read-Host "Digite o caminho do diretório"
            if (Test-Path $caminho) {
                $recurse = Read-Host "Buscar recursivamente? (S/N)"
                if ($recurse -eq 'S') {
                    Find-DuplicateFiles -Path $caminho -Recurse
                } else {
                    Find-DuplicateFiles -Path $caminho
                }
            }
            Read-Host "`nPressione Enter para continuar"
        }
        "3" {
            $dias = Read-Host "Remover arquivos com mais de quantos dias? (padrão:
7)"
            if ([string]::IsNullOrWhiteSpace($dias)) { $dias = 7 }
            Clear-TemporaryFiles -DaysOld $dias
            Read-Host "`nPressione Enter para continuar"
        }
        "4" {
```

```
            $caminho = Read-Host "Digite o caminho do diretório"
            if (Test-Path $caminho) {
                Get-DiskUsageReport -Path $caminho
            }
            Read-Host "`nPressione Enter para continuar"
        }
        "5" {
            $origem = Read-Host "Caminho de origem"
            $destino = Read-Host "Caminho de destino"
            if (Test-Path $origem) {
                New-SmartBackup -Source $origem -Destination $destino
            }
            Read-Host "`nPressione Enter para continuar"
        }
    }
  } while ($opcao -ne "6")
}

# Executar menu
Show-FileManagementMenu
```

5.1.2 Gerenciamento de Processos

Listar e Consultar Processos

```
# Get-Process: Obter processos em execução

# Listar todos os processos
Get-Process
ps  # Alias
```

```
# Processo específico
Get-Process -Name notepad
Get-Process -Id 1234

# Múltiplos processos
Get-Process -Name chrome, firefox, edge

# Com wildcard
Get-Process -Name *office*

# Ordenar por uso de CPU
Get-Process | Sort-Object CPU -Descending | Select-Object -First 10

# Ordenar por uso de memória
Get-Process | Sort-Object WorkingSet -Descending | Select-Object -First 10

# Filtrar processos
Get-Process | Where-Object { $_.CPU -gt 100 }
Get-Process | Where-Object { $_.WorkingSet -gt 100MB }

# Propriedades importantes
$processo = Get-Process -Name powershell | Select-Object -First 1
$processo.Id            # ID do processo
$processo.ProcessName   # Nome do processo
$processo.CPU           # Tempo de CPU (segundos)
$processo.WorkingSet    # Memória física (bytes)
$processo.VirtualMemorySize  # Memória virtual
$processo.Threads.Count  # Número de threads
```

$processo.StartTime *# Hora de início*

$processo.Path *# Caminho do executável*

$processo.Company *# Empresa desenvolvedora*

$processo.MainWindowTitle *# Título da janela principal*

 # Informações detalhadas

Get-Process -Name chrome | Format-List *

Get-Process -Name chrome | Get-Member

Iniciar Processos

 # Start-Process: Iniciar novo processo

 # Iniciar aplicação simples

Start-Process notepad

 # Iniciar com argumentos

Start-Process notepad -ArgumentList "C:\teste.txt"

 # Iniciar como administrador

Start-Process powershell -Verb RunAs

 # Iniciar e aguardar conclusão

Start-Process notepad -Wait

 # Iniciar minimizado

Start-Process calc -WindowStyle Minimized

 # Iniciar maximizado

Start-Process explorer -WindowStyle Maximized

 # Iniciar oculto

Start-Process powershell -ArgumentList "-File C:\script.ps1" -WindowStyle Hidden

 # Redirecionar saída

Start-Process ping -ArgumentList "8.8.8.8" -RedirectStandardOutput "resultado.txt" -NoNewWindow

 # Iniciar com credenciais específicas

$cred = Get-Credential

Start-Process notepad -Credential $cred

 # Capturar objeto Process

$proc = Start-Process notepad -PassThru

$proc.Id

$proc.WaitForExit()

 # Exemplos práticos

 # Abrir URL no navegador padrão

Start-Process "https://www.google.com"

 # Abrir arquivo com aplicação padrão

Start-Process "documento.pdf"

 # Executar comando no CMD

Start-Process cmd -ArgumentList "/c ipconfig /all" -WindowStyle Hidden

 # Executar script PowerShell em nova janela

Start-Process powershell -ArgumentList "-File C:\Scripts\monitor.ps1" -WindowStyle Normal

Parar Processos

 # Stop-Process: Encerrar processo

 # Por nome

Stop-Process -Name notepad

 # Por ID

Stop-Process -Id 1234

 # Forçar encerramento

Stop-Process -Name chrome -Force

 # Múltiplos processos

Stop-Process -Name notepad, wordpad

 # Com confirmação

Stop-Process -Name excel -Confirm

 # Simular (WhatIf)

Stop-Process -Name chrome -WhatIf

 # Parar todos os processos com um padrão

Get-Process -Name *office* | Stop-Process -Force

 # Parar processos usando muita CPU

```
Get-Process | Where-Object { $_.CPU -gt 1000 } | Stop-Process -Force

# Parar processos usando muita memória
Get-Process | Where-Object { $_.WorkingSet -gt 500MB } | Stop-Process

# Usar método Kill() do objeto
$processo = Get-Process -Name notepad
$processo.Kill()

# Parar processo com timeout
$processo = Get-Process -Name chrome | Select-Object -First 1
$processo.CloseMainWindow()
Start-Sleep -Seconds 5
if (-not $processo.HasExited) {
  $processo.Kill()
}
```

Aguardar Processos

```
# Wait-Process: Aguardar encerramento

# Aguardar processo terminar
Wait-Process -Name notepad

# Aguardar com timeout
Wait-Process -Name chrome -Timeout 30

# Aguardar múltiplos processos
Wait-Process -Name excel, word
```

 # Aguardar por ID

$proc = Start-Process notepad -PassThru

Wait-Process -Id $proc.Id

 # Exemplo: executar e aguardar

$processo = Start-Process powershell -ArgumentList "-File C:\script.ps1" -PassThru

Wait-Process -Id $processo.Id

Write-Host "Processo finalizado com código: $($processo.ExitCode)"

Monitoramento de Processos

 # Debug-Process: Anexar debugger

Debug-Process -Name notepad

 # Prioridade de processos

$processo = Get-Process -Name notepad | Select-Object -First 1

 # Ver prioridade atual

$processo.PriorityClass

 # Alterar prioridade

$processo.PriorityClass = "High"

 # Valores: Idle, BelowNormal, Normal, AboveNormal, High, RealTime

 # Módulos carregados pelo processo

$processo = Get-Process -Name powershell | Select-Object -First 1

$processo.Modules | Select-Object ModuleName, FileName

 # Threads do processo

```
$processo.Threads | Select-Object Id, ThreadState, TotalProcessorTime
```

Exemplo Completo: Monitor de Processos

```
# Arquivo: MonitorProcessos.ps1

<#
.SYNOPSIS
   Sistema de monitoramento e gerenciamento de processos
#>

# Função para obter top processos
function Get-TopProcesses {
   param(
      [ValidateSet("CPU", "Memory", "Threads")]
      [string]$SortBy = "Memory",
      [int]$Top = 10
   )

   $processos = Get-Process

   switch ($SortBy) {
      "CPU" {
         $sorted = $processos | Sort-Object CPU -Descending
         $label = "CPU Time (s)"
         $property = "CPU"
      }
      "Memory" {
         $sorted = $processos | Sort-Object WorkingSet -Descending
         $label = "Memory (MB)"
```

```
            $property = "WorkingSet"
        }
        "Threads" {
            $sorted = $processos | Sort-Object {$_.Threads.Count} -Descending
            $label = "Threads"
            $property = @{Name="Threads"; Expression={$_.Threads.Count}}
        }
    }

    Write-Host "`nTop $Top Processos por $SortBy:" -ForegroundColor Cyan

    $sorted | Select-Object -First $Top |
        Select-Object Name, Id,
            @{Name=$label; Expression={
                if ($SortBy -eq "Memory") {
                    [math]::Round($_.$property / 1MB, 2)
                } elseif ($SortBy -eq "CPU") {
                    [math]::Round($_.$property, 2)
                } else {
                    $_.Threads.Count
                }
            }} |
        Format-Table -AutoSize
}

# Função para monitorar processo específico
function Watch-Process {
    param(
```

```
    [Parameter(Mandatory=$true)]
    [string]$ProcessName,
    [int]$IntervalSeconds = 5,
    [int]$Duration = 60
)

$inicio = Get-Date
$limiteTempoWrite-Host "`nMonitorando processo: $ProcessName" -ForegroundColor Cyan
Write-Host "Duração: $Duration segundos | Intervalo: $IntervalSeconds segundos`n" -ForegroundColor Yellow

$medicoes = @()

while (((Get-Date) - $inicio).TotalSeconds -lt $Duration) {
    $processo = Get-Process -Name $ProcessName -ErrorAction SilentlyContinue

    if ($processo) {
        $medicao = [PSCustomObject]@{
            Timestamp = Get-Date -Format "HH:mm:ss"
            CPU = [math]::Round($processo.CPU, 2)
            MemoryMB = [math]::Round($processo.WorkingSet / 1MB, 2)
            Threads = $processo.Threads.Count
            Handles = $processo.HandleCount
        }

        $medicoes += $medicao
```

```
        Write-Host "[$($medicao.Timestamp)] CPU: $($medicao.CPU)s | Memória: $($medicao.MemoryMB) MB | Threads: $($medicao.Threads)" -ForegroundColor Green
    } else {
        Write-Host "Processo não está em execução" -ForegroundColor Red
        break
    }

    Start-Sleep -Seconds $IntervalSeconds
  }

  if ($medicoes.Count -gt 0) {
    Write-Host "`n=== ESTATÍSTICAS ===" -ForegroundColor Cyan
    Write-Host "CPU Média: $([math]::Round(($medicoes.CPU | Measure-Object -Average).Average, 2))s" -ForegroundColor Yellow
    Write-Host "CPU Máxima: $([math]::Round(($medicoes.CPU | Measure-Object -Maximum).Maximum, 2))s" -ForegroundColor Yellow
    Write-Host "Memória Média: $([math]::Round(($medicoes.MemoryMB | Measure-Object -Average).Average, 2)) MB" -ForegroundColor Yellow
    Write-Host "Memória Máxima: $([math]::Round(($medicoes.MemoryMB | Measure-Object -Maximum).Maximum, 2)) MB" -ForegroundColor Yellow
  }
}

# Função para encerrar processos problemáticos
function Stop-ProblematicProcesses {
  param(
    [int]$CPUThreshold = 90,
    [int]$MemoryThresholdMB = 1000,
    [switch]$WhatIf
```

```
)

Write-Host "`nBuscando processos problemáticos..." -ForegroundColor Yellow
Write-Host "Limites: CPU > $CPUThreshold s | Memória > $MemoryThresholdMB MB`n" -ForegroundColor Gray

$processos = Get-Process | Where-Object {
    $_.CPU -gt $CPUThreshold -or ($_.WorkingSet / 1MB) -gt $MemoryThresholdMB
}

if ($processos.Count -eq 0) {
    Write-Host "Nenhum processo problemático encontrado." -ForegroundColor Green
    return
}

Write-Host "Encontrados $($processos.Count) processo(s) problemático(s):" -ForegroundColor Red

foreach ($proc in $processos) {
    $cpu = [math]::Round($proc.CPU, 2)
    $mem = [math]::Round($proc.WorkingSet / 1MB, 2)

    Write-Host "`n[$($proc.Name)] PID: $($proc.Id)" -ForegroundColor Yellow
    Write-Host "  CPU: $cpu s | Memória: $mem MB" -ForegroundColor Cyan

    if (-not $WhatIf) {
        $confirmacao = Read-Host "Encerrar este processo? (S/N)"
```

```
        if ($confirmacao -eq 'S' -or $confirmacao -eq 's') {
            try {
                Stop-Process -Id $proc.Id -Force
                Write-Host "  ✓ Processo encerrado" -ForegroundColor Green
            } catch {
                Write-Host "  ✗ Erro ao encerrar: $_" -ForegroundColor Red
            }
        }
    } else {
        Write-Host "  [WhatIf] Processo seria encerrado" -ForegroundColor Gray
    }
  }
}

# Função para gerar relatório de processos
function Export-ProcessReport {
  param(
    [string]$OutputPath = ".\ProcessReport_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv"
  )

  Write-Host "`nGerando relatório de processos..." -ForegroundColor Yellow

  $processos = Get-Process | Select-Object Name, Id,
    @{Name='Company'; Expression={$_.Company}},
    @{Name='CPU_Seconds'; Expression={[math]::Round($_.CPU, 2)}},
    @{Name='Memory_MB'; Expression={[math]::Round($_.WorkingSet / 1MB, 2)}},
    @{Name='Threads'; Expression={$_.Threads.Count}},
```

```
        @{Name='Handles'; Expression={$_.HandleCount}},
        @{Name='StartTime'; Expression={$_.StartTime}},
        @{Name='Path'; Expression={$_.Path}}

    $processos | Export-Csv -Path $OutputPath -NoTypeInformation

    Write-Host "Relatório salvo em: $OutputPath" -ForegroundColor Green
    Write-Host "Total de processos: $($processos.Count)" -ForegroundColor Cyan
}

# Menu principal
function Show-ProcessMenu {
    do {
        Clear-Host
        Write-Host "╔═══════════════════════════════════════════════════════╗" -ForegroundColor Cyan
        Write-Host "║    GERENCIADOR DE PROCESSOS           ║" -ForegroundColor Cyan
        Write-Host "╚═══════════════════════════════════════════════════════╝" -ForegroundColor Cyan
        Write-Host ""
        Write-Host "1. Top Processos (CPU)" -ForegroundColor White
        Write-Host "2. Top Processos (Memória)" -ForegroundColor White
        Write-Host "3. Monitorar Processo Específico" -ForegroundColor White
        Write-Host "4. Encerrar Processos Problemáticos" -ForegroundColor White
        Write-Host "5. Exportar Relatório" -ForegroundColor White
        Write-Host "6. Sair" -ForegroundColor White
        Write-Host ""
```

```
    $opcao = Read-Host "Escolha uma opção"

    switch ($opcao) {
        "1" {
            Get-TopProcesses -SortBy CPU
            Read-Host "`nPressione Enter para continuar"
        }
        "2" {
            Get-TopProcesses -SortBy Memory
            Read-Host "`nPressione Enter para continuar"
        }
        "3" {
            $nome = Read-Host "Nome do processo"
            $duracao = Read-Host "Duração (segundos)"
            Watch-Process -ProcessName $nome -Duration $duracao
            Read-Host "`nPressione Enter para continuar"
        }
        "4" {
            Stop-ProblematicProcesses
            Read-Host "`nPressione Enter para continuar"
        }
        "5" {
            Export-ProcessReport
            Read-Host "`nPressione Enter para continuar"
        }
    }
} while ($opcao -ne "6")
```

}

 # Executar menu

Show-ProcessMenu

5.1.3 Gerenciamento de Serviços

Listar e Consultar Serviços

 # Get-Service: Obter serviços

 # Listar todos os serviços

Get-Service

 # Serviço específico

Get-Service -Name wuauserv

 # Múltiplos serviços

Get-Service -Name wuauserv, spooler, BITS

 # Com wildcard

Get-Service -Name *audio*

 # Filtrar por status

Get-Service | Where-Object Status -eq 'Running'

Get-Service | Where-Object Status -eq 'Stopped'

 # Serviços com dependências

Get-Service -Name w32time -RequiredServices *# Serviços dos quais depende*

Get-Service -Name w32time -DependentServices *# Serviços que dependem dele*

```
# Propriedades importantes
$servico = Get-Service -Name wuauserv
$servico.Name          # Nome do serviço
$servico.DisplayName    # Nome de exibição
$servico.Status        # Status (Running, Stopped, etc)
$servico.StartType      # Tipo de inicialização
$servico.ServiceType    # Tipo de serviço
$servico.CanStop        # Pode ser parado?
$servico.CanPauseAndContinue  # Pode ser pausado?

# Informações detalhadas (WMI/CIM)
Get-CimInstance -ClassName Win32_Service -Filter "Name='wuauserv'" |
  Select-Object Name, DisplayName, State, StartMode, PathName, StartName

# Serviços agrupados por status
Get-Service | Group-Object Status

# Serviços ordenados
Get-Service | Sort-Object DisplayName
```

Controlar Serviços

```
# Start-Service: Iniciar serviço

# Iniciar serviço
Start-Service -Name wuauserv

# Iniciar com passthru
Start-Service -Name spooler -PassThru
```

 # Iniciar e aguardar

Start-Service -Name BITS

Get-Service -Name BITS | Wait-Service -Status Running

 # Stop-Service: Parar serviço

 # Parar serviço

Stop-Service -Name wuauserv

 # Parar forçadamente

Stop-Service -Name spooler -Force

 # Parar sem confirmação

Stop-Service -Name BITS -Force -NoWait

 # Restart-Service: Reiniciar serviço

 # Reiniciar serviço

Restart-Service -Name wuauserv

 # Reiniciar forçadamente

Restart-Service -Name spooler -Force

 # Suspend-Service e Resume-Service: Pausar/Retomar

 # Pausar serviço (se suportado)

Suspend-Service -Name wuauserv

 # Retomar serviço

Resume-Service -Name wuauserv

 # Set-Service: Modificar propriedades

 # Alterar tipo de inicialização

Set-Service -Name wuauserv -StartupType Automatic

Set-Service -Name wuauserv -StartupType Manual

Set-Service -Name wuauserv -StartupType Disabled

 # Alterar descrição

Set-Service -Name MeuServico -Description "Descrição personalizada"

 # Alterar DisplayName

Set-Service -Name MeuServico -DisplayName "Meu Serviço Custom"

 # Alterar credenciais de execução

$cred = Get-Credential

Set-Service -Name MeuServico -Credential $cred

Criar e Remover Serviços

 # New-Service: Criar novo serviço

 # Criar serviço simples

New-Service -Name "MeuServico" `

  -BinaryPathName "C:\Scripts\meuservico.exe" `

  -DisplayName "Meu Serviço" `

  -Description "Descrição do serviço" `

  -StartupType Automatic

```
# Criar com dependências
New-Service -Name "MeuServico" `
  -BinaryPathName "C:\Scripts\servico.exe" `
  -DependsOn "EventLog", "PlugPlay"

# Criar com credenciais específicas
New-Service -Name "MeuServico" `
  -BinaryPathName "C:\Scripts\servico.exe" `
  -Credential (Get-Credential)

# Remove-Service: Remover serviço (PS 6.0+)
Remove-Service -Name "MeuServico"

# Em versões antigas, usar sc.exe
sc.exe delete MeuServico
```

Exemplo Completo: Gerenciador de Serviços

```
# Arquivo: GerenciadorServicos.ps1

<#
.SYNOPSIS
  Sistema completo de gerenciamento de serviços
#>

# Função para obter status de serviços críticos
function Get-CriticalServicesStatus {
  $servicosCriticos = @(
    "wuauserv",     # Windows Update
```

```
    "BITS",         # Background Intelligent Transfer
    "Spooler",      # Print Spooler
    "EventLog",     # Event Log
    "WinRM",        # Windows Remote Management
    "W32Time",      # Windows Time
    "Dnscache",     # DNS Client
    "LanmanServer",  # Server
    "LanmanWorkstation"  # Workstation
)

Write-Host "`n=== STATUS DE SERVIÇOS CRÍTICOS ===" -ForegroundColor Cyan

$status = foreach ($servico in $servicosCriticos) {
    $svc = Get-Service -Name $servico -ErrorAction SilentlyContinue

    if ($svc) {
        [PSCustomObject]@{
            Nome = $svc.DisplayName
            Status = $svc.Status
            Inicialização = $svc.StartType
            Alerta = if ($svc.Status -ne 'Running' -and $svc.StartType -eq 'Automatic') {
" ⚠️ " } else { "✓" }
        }
    }
}

$status | Format-Table -AutoSize
```

```
    $alertas = $status | Where-Object Alerta -eq " ⚠️ "
    if ($alertas) {
        Write-Host "`nATENÇÃO: $($alertas.Count) serviço(s) com problema!" -ForegroundColor Red
    } else {
        Write-Host "`nTodos os serviços críticos estão OK!" -ForegroundColor Green
    }
}

# Função para iniciar serviços parados (automáticos)
function Start-StoppedAutomaticServices {
    param([switch]$WhatIf)

    Write-Host "`nBuscando serviços automáticos parados..." -ForegroundColor Yellow

    $servicosParados = Get-Service | Where-Object {
        $_.Status -eq 'Stopped' -and $_.StartType -eq 'Automatic'
    }

    if ($servicosParados.Count -eq 0) {
        Write-Host "Todos os serviços automáticos estão em execução." -ForegroundColor Green
        return
    }

    Write-Host "Encontrados $($servicosParados.Count) serviço(s) parado(s):`n" -ForegroundColor Cyan
```

```
    foreach ($servico in $servicosParados) {
        Write-Host "[$($servico.DisplayName)]" -ForegroundColor Yellow

        if ($WhatIf) {
            Write-Host "  [WhatIf] Seria iniciado" -ForegroundColor Gray
        } else {
            try {
                Start-Service -Name $servico.Name -ErrorAction Stop
                Write-Host "  ✓ Iniciado com sucesso" -ForegroundColor Green
            } catch {
                Write-Host "  ✗ Erro: $_" -ForegroundColor Red
            }
        }
    }
}

# Função para monitorar serviço
function Watch-ServiceStatus {
    param(
        [Parameter(Mandatory=$true)]
        [string]$ServiceName,
        [int]$IntervalSeconds = 5,
        [int]$Duration = 60
    )

    $inicio = Get-Date

    Write-Host "`nMonitorando serviço: $ServiceName" -ForegroundColor Cyan
```

```
  Write-Host "Duração: $Duration segundos | Intervalo: $IntervalSeconds segundos`n" -ForegroundColor Yellow

  $mudancas = @()
  $statusAnterior = $null

  while (((Get-Date) - $inicio).TotalSeconds -lt $Duration) {
    $servico = Get-Service -Name $ServiceName -ErrorAction SilentlyContinue

    if ($servico) {
      $timestamp = Get-Date -Format "HH:mm:ss"

      if ($statusAnterior -and $servico.Status -ne $statusAnterior) {
        $mudanca = [PSCustomObject]@{
          Timestamp = $timestamp
          StatusAnterior = $statusAnterior
          StatusNovo = $servico.Status
        }
        $mudancas += $mudanca

        Write-Host "[$timestamp] MUDANÇA: $statusAnterior → $($servico.Status)" -ForegroundColor Red
      } else {
        $cor = if ($servico.Status -eq 'Running') { 'Green' } else { 'Yellow' }
        Write-Host "[$timestamp] Status: $($servico.Status)" -ForegroundColor $cor
      }

      $statusAnterior = $servico.Status
```

```
    } else {
      Write-Host "Serviço não encontrado: $ServiceName" -ForegroundColor Red
      break
    }

    Start-Sleep -Seconds $IntervalSeconds
  }

  if ($mudancas.Count -gt 0) {
    Write-Host "`n=== MUDANÇAS DETECTADAS ===" -ForegroundColor Cyan
    $mudancas | Format-Table -AutoSize
  } else {
    Write-Host "`nNenhuma mudança de status detectada." -ForegroundColor Green
  }
}

# Função para restart seguro de serviços
function Restart-ServiceSafely {
  param(
    [Parameter(Mandatory=$true)]
    [string]$ServiceName,
    [int]$TimeoutSeconds = 30
  )

  Write-Host "`nReiniciando serviço: $ServiceName" -ForegroundColor Yellow

  $servico = Get-Service -Name $ServiceName -ErrorAction SilentlyContinue
```

```
    if (-not $servico) {
        Write-Host "Serviço não encontrado!" -ForegroundColor Red
        return $false
    }

    Write-Host "Status atual: $($servico.Status)" -ForegroundColor Cyan

    # Verificar dependências
    $dependentes = Get-Service -Name $ServiceName -DependentServices |
                   Where-Object Status -eq 'Running'

    if ($dependentes) {
        Write-Host "`nAVISO: Os seguintes serviços dependem de $ServiceName :" -
ForegroundColor Yellow
        $dependentes | ForEach-Object { Write-Host "  - $($_.DisplayName)" -
ForegroundColor Gray }

        $confirmacao = Read-Host "`nContinuar? (S/N)"
        if ($confirmacao -ne 'S' -and $confirmacao -ne 's') {
            Write-Host "Operação cancelada." -ForegroundColor Yellow
            return $false
        }
    }

    try {
        # Parar serviço
        Write-Host "Parando serviço..." -ForegroundColor Yellow
        Stop-Service -Name $ServiceName -Force -ErrorAction Stop
```

```
        # Aguardar parar
        $servico.WaitForStatus('Stopped',
[TimeSpan]::FromSeconds($TimeoutSeconds))
        Write-Host "✓ Serviço parado" -ForegroundColor Green

        # Aguardar um pouco
        Start-Sleep -Seconds 2

        # Iniciar serviço
        Write-Host "Iniciando serviço..." -ForegroundColor Yellow
        Start-Service -Name $ServiceName -ErrorAction Stop

        # Aguardar iniciar
        $servico.WaitForStatus('Running',
[TimeSpan]::FromSeconds($TimeoutSeconds))
        Write-Host "✓ Serviço iniciado" -ForegroundColor Green

        # Verificar status final
        $servico.Refresh()
        Write-Host "`nStatus final: $($servico.Status)" -ForegroundColor Cyan

        return $true
    }
    catch {
        Write-Host "✗ Erro ao reiniciar serviço: $_" -ForegroundColor Red
        return $false
    }
}
```

```
# Função para exportar configuração de serviços
function Export-ServicesConfiguration {
  param(
    [string]$OutputPath = ".\ServicesConfig_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv"
  )

  Write-Host "`nExportando configuração de serviços..." -ForegroundColor Yellow

  $servicos = Get-CimInstance -ClassName Win32_Service |
    Select-Object Name, DisplayName, State, StartMode, PathName, StartName, Description

  $servicos | Export-Csv -Path $OutputPath -NoTypeInformation

  Write-Host "Configuração exportada para: $OutputPath" -ForegroundColor Green
  Write-Host "Total de serviços: $($servicos.Count)" -ForegroundColor Cyan
}

# Função para comparar configurações de serviços
function Compare-ServicesConfiguration {
  param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$BaselineFile
  )
```

```
  Write-Host "`nComparando configuração atual com baseline..." -
ForegroundColor Yellow

  # Carregar baseline
  $baseline = Import-Csv -Path $BaselineFile

  # Obter configuração atual
  $atual = Get-CimInstance -ClassName Win32_Service |
    Select-Object Name, DisplayName, State, StartMode

  $diferencas = @()

  foreach ($svcBaseline in $baseline) {
    $svcAtual = $atual | Where-Object Name -eq $svcBaseline.Name

    if (-not $svcAtual) {
      $diferencas += [PSCustomObject]@{
        Servico = $svcBaseline.Name
        Tipo = "Removido"
        Baseline = "Existia"
        Atual = "Não existe"
      }
    }
    elseif ($svcAtual.State -ne $svcBaseline.State) {
      $diferencas += [PSCustomObject]@{
        Servico = $svcBaseline.DisplayName
        Tipo = "Status"
        Baseline = $svcBaseline.State
```

```
            Atual = $svcAtual.State
        }
    }
    elseif ($svcAtual.StartMode -ne $svcBaseline.StartMode) {
        $diferencas += [PSCustomObject]@{
            Servico = $svcBaseline.DisplayName
            Tipo = "Inicialização"
            Baseline = $svcBaseline.StartMode
            Atual = $svcAtual.StartMode
        }
    }
}

# Verificar novos serviços
foreach ($svcAtual in $atual) {
    if (-not ($baseline | Where-Object Name -eq $svcAtual.Name)) {
        $diferencas += [PSCustomObject]@{
            Servico = $svcAtual.Name
            Tipo = "Novo"
            Baseline = "Não existia"
            Atual = "Existe"
        }
    }
}

if ($diferencas.Count -eq 0) {
    Write-Host "Nenhuma diferença encontrada!" -ForegroundColor Green
} else {
```

```
        Write-Host "`nEncontradas $($diferencas.Count) diferença(s):" -ForegroundColor Red
        $diferencas | Format-Table -AutoSize
    }

    return $diferencas
}

# Menu principal
function Show-ServiceMenu {
    do {
        Clear-Host
        Write-Host "╔═══════════════════════════════════════════════════════════╗" -ForegroundColor Cyan
        Write-Host "║   GERENCIADOR DE SERVIÇOS                                 ║" -ForegroundColor Cyan
        Write-Host "╚═══════════════════════════════════════════════════════════╝" -ForegroundColor Cyan
        Write-Host ""
        Write-Host "1. Status de Serviços Críticos" -ForegroundColor White
        Write-Host "2. Iniciar Serviços Automáticos Parados" -ForegroundColor White
        Write-Host "3. Monitorar Serviço" -ForegroundColor White
        Write-Host "4. Reiniciar Serviço com Segurança" -ForegroundColor White
        Write-Host "5. Exportar Configuração" -ForegroundColor White
        Write-Host "6. Comparar com Baseline" -ForegroundColor White
        Write-Host "7. Sair" -ForegroundColor White
        Write-Host ""

        $opcao = Read-Host "Escolha uma opção"
```

```
switch ($opcao) {
    "1" {
        Get-CriticalServicesStatus
        Read-Host "`nPressione Enter para continuar"
    }
    "2" {
        Start-StoppedAutomaticServices
        Read-Host "`nPressione Enter para continuar"
    }
    "3" {
        $nome = Read-Host "Nome do serviço"
        $duracao = Read-Host "Duração (segundos)"
        Watch-ServiceStatus -ServiceName $nome -Duration $duracao
        Read-Host "`nPressione Enter para continuar"
    }
    "4" {
        $nome = Read-Host "Nome do serviço"
        Restart-ServiceSafely -ServiceName $nome
        Read-Host "`nPressione Enter para continuar"
    }
    "5" {
        Export-ServicesConfiguration
        Read-Host "`nPressione Enter para continuar"
    }
    "6" {
        $arquivo = Read-Host "Caminho do arquivo baseline"
        if (Test-Path $arquivo) {
```

```
            Compare-ServicesConfiguration -BaselineFile $arquivo
        } else {
            Write-Host "Arquivo não encontrado!" -ForegroundColor Red
        }
        Read-Host "`nPressione Enter para continuar"
      }
    }
  } while ($opcao -ne "7")
}

# Executar menu
Show-ServiceMenu
```

5.2 Acesso Remoto e Gerenciamento de Múltiplos Computadores

5.2.1 PowerShell Remoting - Fundamentos

O que é PowerShell Remoting?

PowerShell Remoting permite executar comandos e scripts em computadores remotos de forma segura e eficiente. Baseia-se no protocolo **WS-Management** (Web Services for Management) e usa **WinRM** (Windows Remote Management).

Características:

- Comunicação criptografada por padrão
- Autenticação integrada ao Windows
- Suporte a sessões persistentes
- Execução em múltiplos computadores simultaneamente
- Funciona através de firewalls corporativos

Portas utilizadas:

- **5985**: HTTP (WinRM)
- **5986**: HTTPS (WinRM-HTTPS)

Habilitar PowerShell Remoting

 # No computador de destino (servidor)

 # Executar como Administrador

 # Habilitar Remoting (configuração rápida)

Enable-PSRemoting -Force

 # O que o comando faz:

 # 1. Inicia o serviço WinRM

 # 2. Define WinRM para inicialização automática

 # 3. Cria regras de firewall

 # 4. Registra configurações de sessão

 # Verificar configuração

Get-PSSessionConfiguration

 # Testar WinRM

Test-WSMan

 # Verificar configuração detalhada

winrm get winrm/config

 # Desabilitar Remoting (se necessário)

Disable-PSRemoting -Force

Configuração de TrustedHosts

Para conexões fora de domínio ou com autenticação NTLM:

 # No computador cliente

 # Executar como Administrador

 # Ver lista atual

Get-Item WSMan:\localhost\Client\TrustedHosts

 # Adicionar computador específico

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "Server01"

 # Adicionar múltiplos computadores

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "Server01,Server02,192.168.1.100"

 # Adicionar todos (não recomendado em produção)

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "*" -Force

 # Adicionar mantendo lista existente

$current = (Get-Item WSMan:\localhost\Client\TrustedHosts).Value

$new = "$current,NewServer"

Set-Item WSMan:\localhost\Client\TrustedHosts -Value $new

Firewall - Regras necessárias

 # Verificar regras de firewall para WinRM

Get-NetFirewallRule -Name "WINRM-HTTP-In-TCP*" | Select-Object Name, Enabled, Profile

 # Habilitar regra manualmente (se necessário)

Enable-NetFirewallRule -Name "WINRM-HTTP-In-TCP"

 # Criar regra customizada

New-NetFirewallRule -Name "WinRM-HTTP" `

  -DisplayName "Windows Remote Management (HTTP-In)" `

  -Direction Inbound `

```
  -LocalPort 5985 `
  -Protocol TCP `
  -Action Allow
```

5.2.2 Executando Comandos Remotamente

Invoke-Command - Execução Remota

```
# Sintaxe básica
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process }

# Com credenciais
$cred = Get-Credential
Invoke-Command -ComputerName Server01 -Credential $cred -ScriptBlock {
  Get-Service
}

# Passar argumentos para o script block
Invoke-Command -ComputerName Server01 -ScriptBlock {
  param($ProcessName)
  Get-Process -Name $ProcessName
} -ArgumentList "powershell"

# Usar variáveis locais no remoto
$serviceName = "wuauserv"
Invoke-Command -ComputerName Server01 -ScriptBlock {
  Get-Service -Name $using:serviceName
}

# Executar em múltiplos computadores
$computadores = "Server01", "Server02", "Server03"
```

```
Invoke-Command -ComputerName $computadores -ScriptBlock {
  $env:COMPUTERNAME
  Get-Service -Name wuauserv | Select-Object Status
}

# Executar script local remotamente
Invoke-Command -ComputerName Server01 -FilePath
"C:\Scripts\Diagnostico.ps1"

# Executar em background (job)
$job = Invoke-Command -ComputerName Server01 -ScriptBlock {
  Get-EventLog -LogName System -Newest 100
} -AsJob

# Receber resultados do job
Receive-Job -Job $job

# Executar com throttle (controlar paralelismo)
Invoke-Command -ComputerName $computadores -ScriptBlock {
  Get-Service
} -ThrottleLimit 5

# Salvar resultados em variável
$resultado = Invoke-Command -ComputerName Server01 -ScriptBlock {
  Get-Process | Where-Object CPU -gt 10
}

# Propriedades adicionadas automaticamente
```

$resultado | Select-Object PSComputerName, Name, CPU

Enter-PSSession - Sessão Interativa

 # Iniciar sessão interativa

Enter-PSSession -ComputerName Server01

 # Com credenciais

$cred = Get-Credential

Enter-PSSession -ComputerName Server01 -Credential $cred

 # Comandos executados estão no contexto remoto

[Server01]: PS C:\> Get-Location

[Server01]: PS C:\> Get-Service

[Server01]: PS C:\> $env:COMPUTERNAME

 # Sair da sessão

[Server01]: PS C:\> Exit-PSSession

 # Ou simplesmente

exit

5.2.3 Gerenciamento de Sessões

New-PSSession - Criar Sessões Persistentes

 # Criar sessão

$session = New-PSSession -ComputerName Server01

 # Com credenciais

$cred = Get-Credential

$session = New-PSSession -ComputerName Server01 -Credential $cred

```
# Criar múltiplas sessões
$sessions = New-PSSession -ComputerName Server01, Server02, Server03

# Ver informações da sessão
$session | Format-List *

# Propriedades importantes
$session.ComputerName
$session.State        # Opened, Closed, Broken
$session.Availability   # Available, Busy
$session.Id

# Get-PSSession - Listar sessões
Get-PSSession

# Filtrar sessões
Get-PSSession -ComputerName Server01
Get-PSSession -State Opened

# Usar sessão existente com Invoke-Command
Invoke-Command -Session $session -ScriptBlock {
  Get-Process
}

# Entrar em sessão existente
Enter-PSSession -Session $session

# Remove-PSSession - Fechar sessões
```

```
Remove-PSSession -Session $session

# Fechar todas as sessões
Get-PSSession | Remove-PSSession

# Disconnect-PSSession - Desconectar (manter ativa)
Disconnect-PSSession -Session $session

# Connect-PSSession - Reconectar
Connect-PSSession -Session $session

# Ou reconectar por ID/nome
Get-PSSession -ComputerName Server01 | Connect-PSSession
```

Vantagens de Sessões Persistentes

```
# Sem sessão persistente (cria/destrói conexão a cada vez)
Invoke-Command -ComputerName Server01 -ScriptBlock { $var = 1 }
Invoke-Command -ComputerName Server01 -ScriptBlock { $var }  # null

# Com sessão persistente (mantém estado)
$session = New-PSSession -ComputerName Server01
Invoke-Command -Session $session -ScriptBlock { $var = 1 }
Invoke-Command -Session $session -ScriptBlock { $var }  # 1

# Reusar sessão é mais eficiente
Measure-Command {
  1..10 | ForEach-Object {
    Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Service }
  }
```

```
}

Measure-Command {
  $session = New-PSSession -ComputerName Server01
  1..10 | ForEach-Object {
    Invoke-Command -Session $session -ScriptBlock { Get-Service }
  }
  Remove-PSSession $session
}
```

5.2.4 Gerenciamento de Múltiplos Computadores

Executar em Múltiplos Servidores

```
# Lista de computadores
$computadores = @(
  "Server01"
  "Server02"
  "Server03"
  "192.168.1.100"
)

# Ou ler de arquivo
$computadores = Get-Content "C:\Scripts\servidores.txt"

# Executar comando em todos
$resultados = Invoke-Command -ComputerName $computadores -ScriptBlock {
  [PSCustomObject]@{
    Computador = $env:COMPUTERNAME
    UptimeDias = [math]::Round((Get-Date) - (Get-CimInstance
Win32_OperatingSystem).LastBootUpTime).TotalDays, 2)
```

```
    MemoriaLivreGB = [math]::Round((Get-CimInstance
Win32_OperatingSystem).FreePhysicalMemory / 1MB, 2)
  }
}

$resultados | Format-Table -AutoSize

# Com tratamento de erros
$resultados = foreach ($comp in $computadores) {
  try {
    Invoke-Command -ComputerName $comp -ScriptBlock {
      [PSCustomObject]@{
        Computador = $env:COMPUTERNAME
        Status = "Online"
        SO = (Get-CimInstance Win32_OperatingSystem).Caption
        Erro = $null
      }
    } -ErrorAction Stop
  }
  catch {
    [PSCustomObject]@{
      Computador = $comp
      Status = "Offline"
      SO = $null
      Erro = $_.Exception.Message
    }
  }
}
```

```
$resultados | Format-Table -AutoSize

# Executar em paralelo (mais rápido)
$resultados = Invoke-Command -ComputerName $computadores -ScriptBlock {
    Get-Service -Name wuauserv
} -ThrottleLimit 10

# Com sessões persistentes (ainda mais eficiente)
$sessions = New-PSSession -ComputerName $computadores

$resultados = Invoke-Command -Session $sessions -ScriptBlock {
    Get-HotFix | Select-Object -First 5
}

Remove-PSSession $sessions
```

Exemplo: Inventário de Múltiplos Servidores

```
# Script: Inventario-Servidores.ps1

param(
    [Parameter(Mandatory=$true)]
    [string[]]$ComputerName,

    [PSCredential]$Credential,

    [string]$OutputPath = ".\Inventario_$(Get-Date -Format
'yyyyMMdd_HHmmss').csv"
)
```

```
Write-Host "Coletando inventário de $($ComputerName.Count) servidor(es)..." -
ForegroundColor Cyan


$scriptBlock = {
  try {
    # Sistema Operacional
    $os = Get-CimInstance Win32_OperatingSystem

    # Processador
    $cpu = Get-CimInstance Win32_Processor

    # Memória
    $cs = Get-CimInstance Win32_ComputerSystem

    # Discos
    $discos = Get-CimInstance Win32_LogicalDisk -Filter "DriveType=3"
    $discoInfo = $discos | ForEach-Object {
      "$($_.DeviceID): $([math]::Round($_.FreeSpace/1GB, 2))GB livre de
$([math]::Round($_.Size/1GB, 2))GB"
    }

    # Rede
    $adapters = Get-NetAdapter | Where-Object Status -eq 'Up'
    $ips = $adapters | Get-NetIPAddress -AddressFamily IPv4 | Select-Object -
ExpandProperty IPAddress

    [PSCustomObject]@{
      Computador = $env:COMPUTERNAME
```

```
        SO = $os.Caption
        Versao = $os.Version
        Arquitetura = $os.OSArchitecture
        UltimoInicioMB = $os.LastBootUpTime
        Processador = $cpu.Name
        Nucleos = $cpu.NumberOfCores
        NucleosLogicos = $cpu.NumberOfLogicalProcessors
        MemoriaTotalGB = [math]::Round($cs.TotalPhysicalMemory / 1GB, 2)
        MemoriaLivreGB = [math]::Round($os.FreePhysicalMemory / 1MB, 2)
        Discos = $discoInfo -join " | "
        IPAddresses = $ips -join ", "
        Status = "Sucesso"
        Erro = $null
    }
  }
```

6. Melhores Práticas e Segurança

• Gestão de credenciais e segurança em scripts

• Padronização e documentação

5. ADMINISTRAÇÃO DE SISTEMAS OPERACIONAIS WINDOWS

5.1 Gerenciamento de Arquivos, Processos e Serviços

5.1.1 Gerenciamento de Arquivos e Diretórios

Cmdlets Básicos de Manipulação de Arquivos

```
# Listar arquivos e diretórios
Get-ChildItem              # Lista itens no diretório atual
Get-ChildItem C:\Windows   # Lista itens em diretório específico
Get-ChildItem -Path C:\ -Recurse # Lista recursivamente
Get-ChildItem -File        # Apenas arquivos
Get-ChildItem -Directory   # Apenas diretórios
```

```
Get-ChildItem -Hidden          # Incluir arquivos ocultos
Get-ChildItem -Filter "*.txt"    # Filtrar por padrão

# Aliases comuns
ls       # Alias Unix-like
dir      # Alias DOS-like
gci      # Alias PowerShell
```

Navegação e Localização

```
# Obter localização atual
Get-Location
pwd      # Alias

# Mudar diretório
Set-Location C:\Windows
cd C:\Windows    # Alias

# Voltar ao diretório anterior
Set-Location -Path $OLDPWD
cd -

# Ir para diretório home
Set-Location ~
cd ~

# Criar estrutura de caminhos
$caminho = Join-Path -Path "C:\Logs" -ChildPath "2025\10"
$caminho  # C:\Logs\2025\10
```

 # Dividir caminho

Split-Path "C:\Windows\System32\cmd.exe" -Parent *# C:\Windows\System32*

Split-Path "C:\Windows\System32\cmd.exe" -Leaf *# cmd.exe*

Split-Path "C:\Windows\System32\cmd.exe" -Extension *# .exe*

 # Converter caminho relativo para absoluto

Resolve-Path ".\arquivo.txt"

 # Testar se caminho existe

Test-Path "C:\Windows\System32"

Test-Path "C:\arquivo_inexistente.txt"

 # Testar tipo

Test-Path "C:\Windows" -PathType Container *# Diretório*

Test-Path "C:\Windows\notepad.exe" -PathType Leaf *# Arquivo*

Criar, Copiar, Mover e Excluir

 # Criar novo item

New-Item -Path "C:\Temp\arquivo.txt" -ItemType File

New-Item -Path "C:\Temp\MinhaPasta" -ItemType Directory

 # Criar múltiplos diretórios

New-Item -Path "C:\Logs\2025\10\15" -ItemType Directory -Force

 # Criar arquivo com conteúdo

New-Item -Path "teste.txt" -ItemType File -Value "Conteúdo inicial"

 # Copiar arquivo

Copy-Item -Path "origem.txt" -Destination "destino.txt"

```
# Copiar diretório recursivamente
Copy-Item -Path "C:\Origem" -Destination "D:\Destino" -Recurse

# Copiar múltiplos arquivos
Copy-Item -Path "C:\Logs\*.log" -Destination "D:\Backup\"

# Copiar com filtro
Get-ChildItem -Path "C:\Origem" -Filter "*.txt" |
  Copy-Item -Destination "C:\Destino"

# Mover arquivo
Move-Item -Path "arquivo.txt" -Destination "C:\NovoLocal\"

# Renomear arquivo
Rename-Item -Path "antigo.txt" -NewName "novo.txt"

# Renomear em lote
Get-ChildItem -Filter "*.tmp" |
  Rename-Item -NewName { $_.Name -replace '.tmp','.txt' }

# Excluir arquivo
Remove-Item -Path "arquivo.txt"

# Excluir diretório e conteúdo
Remove-Item -Path "C:\Temp" -Recurse -Force

# Excluir com confirmação
```

```
Remove-Item -Path "importante.txt" -Confirm

# Excluir arquivos antigos
Get-ChildItem -Path "C:\Logs" -Filter "*.log" |
  Where-Object LastWriteTime -lt (Get-Date).AddDays(-30) |
  Remove-Item -Force
```

Propriedades e Atributos de Arquivos

```
# Obter informações detalhadas
$arquivo = Get-Item "C:\Windows\notepad.exe"
$arquivo | Format-List *

# Propriedades importantes
$arquivo.Name           # Nome do arquivo
$arquivo.FullName       # Caminho completo
$arquivo.Length         # Tamanho em bytes
$arquivo.Extension      # Extensão
$arquivo.CreationTime   # Data de criação
$arquivo.LastWriteTime  # Última modificação
$arquivo.LastAccessTime # Último acesso
$arquivo.Attributes     # Atributos

# Modificar atributos
$arquivo.Attributes = "ReadOnly"
$arquivo.Attributes = "Hidden"
$arquivo.Attributes = "Archive, ReadOnly"

# Remover atributos
$arquivo.Attributes = "Normal"
```

```
# Modificar datas
$arquivo.CreationTime = Get-Date "2025-01-01"
$arquivo.LastWriteTime = Get-Date

# Obter hash de arquivo (verificação de integridade)
Get-FileHash "arquivo.zip" -Algorithm SHA256
Get-FileHash "arquivo.zip" -Algorithm MD5

# Comparar hashes
$hash1 = (Get-FileHash "arquivo1.txt").Hash
$hash2 = (Get-FileHash "arquivo2.txt").Hash
if ($hash1 -eq $hash2) {
  Write-Host "Arquivos idênticos"
} else {
  Write-Host "Arquivos diferentes"
}
```

Conteúdo de Arquivos

```
# Ler conteúdo de arquivo
Get-Content "arquivo.txt"
cat "arquivo.txt"   # Alias

# Ler com encoding específico
Get-Content "arquivo.txt" -Encoding UTF8

# Ler primeiras N linhas
Get-Content "arquivo.txt" -TotalCount 10
Get-Content "arquivo.txt" -Head 10
```

```
# Ler últimas N linhas
Get-Content "arquivo.txt" -Tail 10

# Monitorar arquivo (tail -f)
Get-Content "log.txt" -Wait -Tail 10

# Escrever conteúdo (sobrescreve)
Set-Content -Path "arquivo.txt" -Value "Novo conteúdo"

# Adicionar conteúdo (append)
Add-Content -Path "log.txt" -Value "Nova entrada de log"

# Limpar conteúdo (mantém arquivo)
Clear-Content "arquivo.txt"

# Ler arquivo como string única
$conteudo = Get-Content "arquivo.txt" -Raw

# Processar linha por linha
Get-Content "arquivo.txt" | ForEach-Object {
  Write-Host "Linha: $_"
}

# Buscar padrão em arquivo
Select-String -Path "arquivo.txt" -Pattern "erro"

# Buscar em múltiplos arquivos
```

```
Get-ChildItem -Filter "*.log" |
  Select-String -Pattern "error" |
  Select-Object Path, LineNumber, Line
```

Compressão de Arquivos

```
# Criar arquivo ZIP
Compress-Archive -Path "C:\Dados" -DestinationPath "C:\Backup\dados.zip"

# Adicionar a ZIP existente
Compress-Archive -Path "C:\Novos" -DestinationPath "C:\Backup\dados.zip" -Update

# Nível de compressão
Compress-Archive -Path "C:\Dados" -DestinationPath "dados.zip" -CompressionLevel Optimal
# Níveis: NoCompression, Fastest, Optimal

# Extrair arquivo ZIP
Expand-Archive -Path "dados.zip" -DestinationPath "C:\Extraidos"

# Extrair forçando sobrescrita
Expand-Archive -Path "dados.zip" -DestinationPath "C:\Extraidos" -Force

# Listar conteúdo de ZIP
Add-Type -AssemblyName System.IO.Compression.FileSystem
$zip = [System.IO.Compression.ZipFile]::OpenRead("dados.zip")
$zip.Entries | Select-Object Name, Length, CompressedLength
$zip.Dispose()
```

Exemplo Completo: Sistema de Gerenciamento de Arquivos

```
# Arquivo: GerenciadorArquivos.ps1
```

```
<#
.SYNOPSIS
   Sistema completo de gerenciamento de arquivos
#>

# Função para organizar arquivos por extensão
function Organize-FilesByExtension {
   param(
      [Parameter(Mandatory=$true)]
      [ValidateScript({Test-Path $_})]
      [string]$SourcePath,

      [Parameter(Mandatory=$true)]
      [string]$DestinationPath,

      [switch]$WhatIf
   )

   Write-Host "`n=== ORGANIZAÇÃO DE ARQUIVOS ===" -ForegroundColor Cyan
   Write-Host "Origem: $SourcePath" -ForegroundColor Yellow

   # Obter todos os arquivos
   $arquivos = Get-ChildItem -Path $SourcePath -File

   if ($arquivos.Count -eq 0) {
      Write-Host "Nenhum arquivo encontrado." -ForegroundColor Yellow
      return
```

```
}

Write-Host "Encontrados $($arquivos.Count) arquivos" -ForegroundColor Cyan

# Agrupar por extensão
$grupos = $arquivos | Group-Object Extension

foreach ($grupo in $grupos) {
    $extensao = if ($grupo.Name) { $grupo.Name.TrimStart('.') } else { "SemExtensao" }
    $pastaDestino = Join-Path $DestinationPath $extensao

    Write-Host "`nProcessando $($grupo.Count) arquivo(s) .$extensao" -ForegroundColor Yellow

    if (-not $WhatIf) {
        if (-not (Test-Path $pastaDestino)) {
            New-Item -Path $pastaDestino -ItemType Directory | Out-Null
            Write-Host "  Pasta criada: $pastaDestino" -ForegroundColor Green
        }
    }

    foreach ($arquivo in $grupo.Group) {
        $destino = Join-Path $pastaDestino $arquivo.Name

        if ($WhatIf) {
            Write-Host "  [SIMULAÇÃO] Mover: $($arquivo.Name) → $pastaDestino" -ForegroundColor Gray
        } else {
```

```
            try {
                Move-Item -Path $arquivo.FullName -Destination $destino -Force
                Write-Host "  ✓ Movido: $($arquivo.Name)" -ForegroundColor Green
            }
            catch {
                Write-Host "  ✗ Erro ao mover $($arquivo.Name): $_" -ForegroundColor
Red
            }
        }
    }
  }

  if ($WhatIf) {
    Write-Host "`n(Simulação - use sem -WhatIf para executar)" -ForegroundColor
Yellow
  } else {
    Write-Host "`nOrganização concluída!" -ForegroundColor Green
  }
}

# Função para buscar arquivos duplicados
function Find-DuplicateFiles {
  param(
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Path $_})]
    [string]$Path,

    [switch]$Recurse
  )
```

```
  Write-Host "`n=== BUSCA DE ARQUIVOS DUPLICADOS ===" -ForegroundColor Cyan
  Write-Host "Analisando: $Path" -ForegroundColor Yellow

  $parametros = @{
    Path = $Path
    File = $true
  }
  if ($Recurse) { $parametros.Recurse = $true }

  Write-Host "Calculando hashes..." -ForegroundColor Yellow

  $arquivos = Get-ChildItem @parametros | ForEach-Object {
    [PSCustomObject]@{
      Path = $_.FullName
      Name = $_.Name
      Size = $_.Length
      Hash = (Get-FileHash $_.FullName -Algorithm MD5).Hash
    }
  }

  $duplicados = $arquivos | Group-Object Hash | Where-Object Count -gt 1

  if ($duplicados) {
    Write-Host "`nEncontrados $($duplicados.Count) grupos de arquivos duplicados:" -ForegroundColor Red

    foreach ($grupo in $duplicados) {
```

```
        Write-Host "`n--- Grupo (Hash: $($grupo.Name.Substring(0,8))...) ---" -ForegroundColor Yellow
        $primeiroArquivo = $grupo.Group[0]
        Write-Host "Tamanho: $([math]::Round($primeiroArquivo.Size / 1KB, 2)) KB" -ForegroundColor Cyan

        foreach ($arquivo in $grupo.Group) {
            Write-Host "  $($arquivo.Path)" -ForegroundColor White
        }
    }

    # Calcular espaço desperdiçado
    $espacoDuplicado = ($duplicados | ForEach-Object {
        $primeiroArquivo = $_.Group[0]
        $primeiroArquivo.Size * ($_.Count - 1)
    } | Measure-Object -Sum).Sum

    Write-Host "`nEspaço desperdiçado: $([math]::Round($espacoDuplicado / 1MB, 2)) MB" -ForegroundColor Red
  } else {
    Write-Host "`nNenhum arquivo duplicado encontrado." -ForegroundColor Green
  }
}

# Função para análise de uso de disco
function Get-DiskUsageReport {
  param(
    [Parameter(Mandatory=$true)]
```

```
    [ValidateScript({Test-Path $_})]
    [string]$Path,

    [int]$TopN = 10
)

Write-Host "` n=== RELATÓRIO DE USO DE DISCO ===" -ForegroundColor Cyan
Write-Host "Analisando: $Path" -ForegroundColor Yellow

# Analisar subdiretórios
$diretorios = Get-ChildItem -Path $Path -Directory -ErrorAction SilentlyContinue

$relatorio = foreach ($dir in $diretorios) {
    Write-Host "." -NoNewline -ForegroundColor Gray

    $tamanho = (Get-ChildItem -Path $dir.FullName -Recurse -File -ErrorAction
SilentlyContinue |
            Measure-Object -Property Length -Sum).Sum

    [PSCustomObject]@{
        Diretorio = $dir.Name
        CaminhoCompleto = $dir.FullName
        TamanhoBytes = $tamanho
        TamanhoMB = [math]::Round($tamanho / 1MB, 2)
        TamanhoGB = [math]::Round($tamanho / 1GB, 2)
        Arquivos = (Get-ChildItem -Path $dir.FullName -Recurse -File -ErrorAction
SilentlyContinue).Count
    }
}
```

```
    Write-Host "`n"

    # Top N maiores diretórios
    $top = $relatorio | Sort-Object TamanhoBytes -Descending | Select-Object -First $TopN

    Write-Host "TOP $TopN MAIORES DIRETÓRIOS:" -ForegroundColor Yellow
    $top | Format-Table Diretorio, TamanhoGB, TamanhoMB, Arquivos -AutoSize

    # Total
    $total = ($relatorio | Measure-Object -Property TamanhoBytes -Sum).Sum
    Write-Host "TOTAL: $([math]::Round($total / 1GB, 2)) GB" -ForegroundColor Cyan
}

# Função para backup incremental
function Start-IncrementalBackup {
    param(
        [Parameter(Mandatory=$true)]
        [ValidateScript({Test-Path $_})]
        [string]$SourcePath,

        [Parameter(Mandatory=$true)]
        [string]$BackupPath,

        [int]$DaysModified = 1
    )
```

```
  Write-Host "`n=== BACKUP INCREMENTAL ===" -ForegroundColor Cyan

  # Criar diretório de backup
  $timestamp = Get-Date -Format "yyyyMMdd_HHmmss"
  $backupDestino = Join-Path $BackupPath "Backup_$timestamp"
  New-Item -Path $backupDestino -ItemType Directory -Force | Out-Null

  Write-Host "Origem: $SourcePath" -ForegroundColor Yellow
  Write-Host "Destino: $backupDestino" -ForegroundColor Yellow

  # Buscar arquivos modificados
  $dataLimite = (Get-Date).AddDays(-$DaysModified)
  $arquivos = Get-ChildItem -Path $SourcePath -Recurse -File |
      Where-Object LastWriteTime -gt $dataLimite

  if ($arquivos.Count -eq 0) {
    Write-Host "Nenhum arquivo modificado nos últimos $DaysModified dia(s)." -
ForegroundColor Yellow
    return
  }

  Write-Host "Arquivos a fazer backup: $($arquivos.Count)" -ForegroundColor
Cyan

  $contador = 0
  foreach ($arquivo in $arquivos) {
    $contador++
    Write-Progress -Activity "Backup em andamento" -Status "$contador de
$($arquivos.Count)" -PercentComplete (($contador / $arquivos.Count) * 100)
```

```
    # Recriar estrutura de diretórios
    $relativePath = $arquivo.FullName.Replace($SourcePath, '')
    $destinoArquivo = Join-Path $backupDestino $relativePath
    $pastaPai = Split-Path $destinoArquivo -Parent

    if (-not (Test-Path $pastaPai)) {
        New-Item -Path $pastaPai -ItemType Directory -Force | Out-Null
    }

    Copy-Item -Path $arquivo.FullName -Destination $destinoArquivo -Force
}

Write-Progress -Activity "Backup em andamento" -Completed

# Comprimir backup
Write-Host "Comprimindo backup..." -ForegroundColor Yellow
$zipPath = "$backupDestino.zip"
Compress-Archive -Path $backupDestino -DestinationPath $zipPath -
CompressionLevel Optimal

# Remover pasta temporária
Remove-Item -Path $backupDestino -Recurse -Force

$zipInfo = Get-Item $zipPath
Write-Host "`n✓ Backup concluído!" -ForegroundColor Green
Write-Host "Arquivo: $zipPath" -ForegroundColor Cyan
Write-Host "Tamanho: $([math]::Round($zipInfo.Length / 1MB, 2)) MB" -
ForegroundColor Cyan
```

}

5.1.2 Gerenciamento de Processos

Listar e Analisar Processos

```
# Listar todos os processos
Get-Process

# Processo específico
Get-Process -Name "notepad"
Get-Process -Id 1234

# Múltiplos processos
Get-Process -Name "chrome", "firefox", "edge"

# Propriedades detalhadas
Get-Process -Name "powershell" | Format-List *

# Propriedades importantes
$proc = Get-Process -Name "powershell" | Select-Object -First 1
$proc.Name          # Nome do processo
$proc.Id            # PID
$proc.CPU           # Tempo de CPU (segundos)
$proc.WorkingSet    # Memória física (bytes)
$proc.VirtualMemorySize # Memória virtual
$proc.Threads.Count  # Número de threads
$proc.StartTime     # Hora de início
$proc.Path          # Caminho do executável
$proc.Company       # Nome da empresa
$proc.ProductVersion  # Versão do produto
```

 # Filtrar processos

Get-Process | Where-Object CPU -gt 10

Get-Process | Where-Object WorkingSet -gt 100MB

Get-Process | Where-Object Company -like "*Microsoft*"

 # Ordenar por uso de recursos

Get-Process | Sort-Object CPU -Descending | Select-Object -First 10

Get-Process | Sort-Object WorkingSet -Descending | Select-Object -First 10

 # Agrupar por empresa

Get-Process | Group-Object Company | Sort-Object Count -Descending

 # Estatísticas

Get-Process | Measure-Object -Property CPU, WorkingSet -Sum -Average

 # Exportar lista de processos

Get-Process | Export-Csv "processos_$(Get-Date -Format 'yyyyMMdd_HHmmss').csv" -NoTypeInformation

Iniciar e Parar Processos

 # Iniciar processo

Start-Process "notepad.exe"

 # Iniciar com argumentos

Start-Process "notepad.exe" -ArgumentList "C:\arquivo.txt"

 # Iniciar e aguardar conclusão

Start-Process "ping.exe" -ArgumentList "google.com" -Wait

```
# Iniciar como administrador
Start-Process "powershell.exe" -Verb RunAs

# Iniciar oculto
Start-Process "cmd.exe" -WindowStyle Hidden

# Iniciar e capturar objeto do processo
$proc = Start-Process "notepad.exe" -PassThru
$proc.Id

# Parar processo por nome
Stop-Process -Name "notepad"

# Parar processo por ID
Stop-Process -Id 1234

# Parar forçadamente
Stop-Process -Name "chrome" -Force

# Parar com confirmação
Stop-Process -Name "excel" -Confirm

# Parar todos os processos com nome específico
Get-Process -Name "chrome" | Stop-Process -Force

# Aguardar processo terminar
$proc = Get-Process -Name "setup"
```

```
$proc.WaitForExit()

# Aguardar com timeout
$proc.WaitForExit(30000)  # 30 segundos
```

Monitoramento Avançado de Processos

```
# Obter processos com proprietário (requer privilégios)
Get-CimInstance Win32_Process | Select-Object ProcessId, Name, @{
  Name='Owner'
  Expression={
    $owner = $_.GetOwner()
    "$($owner.Domain)\$($owner.User)"
  }
}

# Obter linha de comando completa
Get-CimInstance Win32_Process |
  Where-Object Name -eq "powershell.exe" |
  Select-Object ProcessId, Name, CommandLine

# Informações detalhadas de memória
$proc = Get-Process -Name "chrome" | Select-Object -First 1
[PSCustomObject]@{
  Nome = $proc.Name
  PID = $proc.Id
  'WorkingSet (MB)' = [math]::Round($proc.WorkingSet / 1MB, 2)
  'PrivateMemory (MB)' = [math]::Round($proc.PrivateMemorySize64 / 1MB, 2)
  'VirtualMemory (MB)' = [math]::Round($proc.VirtualMemorySize64 / 1MB, 2)
  'PeakWorkingSet (MB)' = [math]::Round($proc.PeakWorkingSet64 / 1MB, 2)
```

```
}

# Monitorar alterações em processos
$antes = Get-Process
Start-Sleep -Seconds 10
$depois = Get-Process

# Novos processos
Compare-Object $antes $depois -Property Name, Id |
  Where-Object SideIndicator -eq '=>' |
  Select-Object Name, Id

# Processos encerrados
Compare-Object $antes $depois -Property Name, Id |
  Where-Object SideIndicator -eq '<=' |
  Select-Object Name, Id
```

Exemplo Completo: Monitor de Processos

```
# Arquivo: MonitorProcessos.ps1

<#
.SYNOPSIS
  Monitora processos em tempo real
#>

param(
  [int]$CPUThreshold = 80,
  [int]$MemoryMBThreshold = 500,
  [int]$IntervalSeconds = 5,
```

```
    [switch]$ContinuousMonitoring
)

function Get-ProcessMetrics {
    param([int]$CPUThreshold, [int]$MemoryMBThreshold)

    $processos = Get-Process | Where-Object {
        ($_.CPU -gt $CPUThreshold) -or
        (($_.WorkingSet / 1MB) -gt $MemoryMBThreshold)
    }

    $metricas = foreach ($proc in $processos) {
        [PSCustomObject]@{
            Nome = $proc.Name
            PID = $proc.Id
            'CPU (s)' = [math]::Round($proc.CPU, 2)
            'Memória (MB)' = [math]::Round($proc.WorkingSet / 1MB, 2)
            Threads = $proc.Threads.Count
            Empresa = $proc.Company
            Caminho = $proc.Path
            Status = if ($proc.CPU -gt $CPUThreshold) { "⚠️ CPU ALTA" } else { "⚠️
MEMÓRIA ALTA" }
        }
    }

    return $metricas
}
```

```
function Show-ProcessDashboard {
    param($Metricas)

    Clear-Host
    Write-Host "╔═══════════════════════════════════════════════════════════════════════════════╗" -ForegroundColor Cyan
    Write-Host "║          MONITOR DE PROCESSOS EM TEMPO REAL                                   ║" -ForegroundColor Cyan
    Write-Host "╚═══════════════════════════════════════════════════════════════════════════════╝" -ForegroundColor Cyan
    Write-Host ""
    Write-Host "Data/Hora: $(Get-Date -Format 'dd/MM/yyyy HH:mm:ss')" -ForegroundColor Yellow
    Write-Host "Limites: CPU > $CPUThreshold s | Memória > $MemoryMBThreshold MB" -ForegroundColor Yellow
    Write-Host ""

    if ($Metricas.Count -eq 0) {
        Write-Host "✓ Nenhum processo acima dos limites" -ForegroundColor Green
    } else {
        Write-Host " ⚠️  $($Metricas.Count) PROCESSO(S) ACIMA DOS LIMITES" -ForegroundColor Red
        Write-Host ""
        $Metricas | Format-Table Nome, PID, 'CPU (s)', 'Memória (MB)', Threads, Status -AutoSize
    }

    # Estatísticas gerais
    $todosProcessos = Get-Process
```

```
    $cpuTotal = ($todosProcessos | Measure-Object -Property CPU -Sum).Sum
    $memoriaTotal = ($todosProcessos | Measure-Object -Property WorkingSet -Sum).Sum / 1GB

    Write-Host "`n--- ESTATÍSTICAS GERAIS ---" -ForegroundColor Cyan
    Write-Host "Total de Processos: $($todosProcessos.Count)" -ForegroundColor White
    Write-Host "CPU Total: $([math]::Round($cpuTotal, 2)) s" -ForegroundColor White
    Write-Host "Memória Total: $([math]::Round($memoriaTotal, 2)) GB" -ForegroundColor White
}

# Loop de monitoramento
do {
    $metricas = Get-ProcessMetrics -CPUThreshold $CPUThreshold -MemoryMBThreshold $MemoryMBThreshold
    Show-ProcessDashboard -Metricas $metricas

    if ($ContinuousMonitoring) {
        Write-Host "`nAtualizando em $IntervalSeconds segundos... (Ctrl+C para sair)" -ForegroundColor Gray
        Start-Sleep -Seconds $IntervalSeconds
    }
} while ($ContinuousMonitoring)
```

5.1.3 Gerenciamento de Serviços

Listar e Consultar Serviços

```
# Listar todos os serviços
Get-Service
```

```
# Serviço específico
Get-Service -Name "wuauserv"

# Múltiplos serviços
Get-Service -Name "wuauserv", "BITS", "Spooler"

# Filtrar por status
Get-Service | Where-Object Status -eq "Running"
Get-Service | Where-Object Status -eq "Stopped"

# Filtrar por tipo de inicialização
Get-Service | Where-Object StartType -eq "Automatic"
Get-Service | Where-Object StartType -eq "Manual"
Get-Service | Where-Object StartType -eq "Disabled"

# Propriedades detalhadas
$servico = Get-Service -Name "wuauserv"
$servico | Format-List *

$servico.Name         # Nome do serviço
$servico.DisplayName  # Nome de exibição
$servico.Status       # Status (Running, Stopped, etc)
$servico.StartType    # Tipo de inicialização
$servico.DependentServices    # Serviços que dependem deste
$servico.ServicesDependedOn   # Serviços dos quais este depende

# Buscar serviços por nome de exibição
Get-Service | Where-Object DisplayName -like "*Update*"
```

 # Ordenar serviços

Get-Service | Sort-Object Status, DisplayName

 # Agrupar por status

Get-Service | Group-Object Status | Select-Object Name, Count

 # Exportar lista

Get-Service | Export-Csv "servicos.csv" -NoTypeInformation

Iniciar, Parar e Gerenciar Serviços

 # Iniciar serviço

Start-Service -Name "wuauserv"

 # Parar serviço

Stop-Service -Name "wuauserv"

 # Reiniciar serviço

Restart-Service -Name "wuauserv"

 # Suspender serviço (se suportado)

Suspend-Service -Name "servicoX"

 # Retomar serviço

Resume-Service -Name "servicoX"

 # Múltiplos serviços

Start-Service -Name "BITS", "wuauserv"

Stop-Service -Name "BITS", "wuauserv"

 # Forçar parada (incluir dependentes)

Stop-Service -Name "wuauserv" -Force

 # Aguardar serviço iniciar

$servico = Get-Service -Name "wuauserv"

$servico.WaitForStatus("Running", (New-TimeSpan -Seconds 30))

 # Configurar tipo de inicialização

Set-Service -Name "wuauserv" -StartupType Automatic

Set-Service -Name "wuauserv" -StartupType Manual

Set-Service -Name "wuauserv" -StartupType Disabled

 # Alterar descrição do serviço (requer WMI/CIM)

$servico = Get-CimInstance -ClassName Win32_Service -Filter "Name='wuauserv'"

$servico | Set-CimInstance -Property @{Description="Serviço de Atualização do Windows"}

 # Modificar conta de logon do serviço

$servico = Get-CimInstance -ClassName Win32_Service -Filter "Name='MeuServico'"

$servico | Invoke-CimMethod -MethodName Change -Arguments @{

  StartName = "DOMINIO\Usuario"

  StartPassword = "senha"

}

Dependências de Serviços

 # Obter dependências

$servico = Get-Service -Name "wuauserv"

```
# Serviços dos quais depende (pré-requisitos)
$servico.ServicesDependedOn | Select-Object Name, DisplayName, Status

# Serviços que dependem deste
$servico.DependentServices | Select-Object Name, DisplayName, Status

# Verificar dependências antes de parar
function Stop-ServiceWithDependencies {
  param([string]$ServiceName)

  $servico = Get-Service -Name $ServiceName

  if ($servico.DependentServices.Count -gt 0) {
    Write-Host "Serviços dependentes:" -ForegroundColor Yellow
    $servico.DependentServices | ForEach-Object {
      Write-Host "  - $($_.DisplayName) ($($_.Status))" -ForegroundColor Cyan
    }

    $confirmar = Read-Host "Deseja parar todos os serviços dependentes? (S/N)"
    if ($confirmar -eq 'S') {
      Stop-Service -Name $ServiceName -Force
    }
  } else {
    Stop-Service -Name $ServiceName
  }
}
```

```
# Iniciar serviço e suas dependências
function Start-ServiceWithDependencies {
  param([string]$ServiceName)

  $servico = Get-Service -Name $ServiceName

  # Iniciar dependências primeiro
  foreach ($dep in $servico.ServicesDependedOn) {
    if ($dep.Status -ne 'Running') {
      Write-Host "Iniciando dependência: $($dep.DisplayName)" -
ForegroundColor Yellow
      Start-Service -Name $dep.Name
    }
  }

  # Iniciar serviço principal
  Write-Host "Iniciando serviço: $($servico.DisplayName)" -ForegroundColor
Green
  Start-Service -Name $ServiceName
}
```

Exemplo Completo: Gerenciador de Serviços

```
# Arquivo: GerenciadorServicos.ps1

<#
.SYNOPSIS
  Gerenciador completo de serviços do Windows
#>

# Função para obter serviços críticos
```

```
function Get-CriticalServices {
  $servicosCriticos = @(
    "wuauserv",    # Windows Update
    "BITS",        # Serviço de Transferência Inteligente
    "EventLog",    # Log de Eventos
    "WinRM",       # Windows Remote Management
    "W32Time",     # Hora do Windows
    "Spooler",     # Spooler de Impressão
    "Dhcp",        # Cliente DHCP
    "Dnscache",    # Cliente DNS
    "LanmanServer", # Servidor
    "LanmanWorkstation" # Estação de Trabalho
  )

  $resultado = foreach ($nome in $servicosCriticos) {
    $servico = Get-Service -Name $nome -ErrorAction SilentlyContinue

    if ($servico) {
      [PSCustomObject]@{
        Nome = $servico.Name
        NomeExibicao = $servico.DisplayName
        Status = $servico.Status
        TipoInicio = $servico.StartType
        Alerta = if ($servico.Status -ne 'Running' -and $servico.StartType -eq
'Automatic') {
          "⚠️ ATENÇÃO"
        } else {
          "✅ OK"
```

```
            }
        }
    }
}

    return $resultado
}

# Função para reiniciar serviços problemáticos
function Restart-ProblematicServices {
    param(
        [int]$UptimeHoursThreshold = 720  # 30 dias
    )

    Write-Host "Verificando serviços problemáticos..." -ForegroundColor Yellow

    $servicosProblematicos = Get-CimInstance Win32_Service | Where-Object {
        $_.State -eq 'Running' -and $_.StartMode -eq 'Auto'
    } | ForEach-Object {
        try {
            $processo = Get-Process -Id $_.ProcessId -ErrorAction Stop
            $uptime = (Get-Date) - $processo.StartTime

            if ($uptime.TotalHours -gt $UptimeHoursThreshold) {
                [PSCustomObject]@{
                    Nome = $_.Name
                    DisplayName = $_.DisplayName
                    UptimeHoras = [math]::Round($uptime.TotalHours, 2)
```

```
            PID = $_.ProcessId
          }
        }
      }
      catch {
        # Processo não encontrado ou sem permissão
      }
    }

    if ($servicosProblematicos) {
      Write-Host "`nServiços com uptime alto:" -ForegroundColor Red
      $servicosProblematicos | Format-Table -AutoSize

      $confirmar = Read-Host "`nDeseja reiniciar estes serviços? (S/N)"
      if ($confirmar -eq 'S') {
        foreach ($servico in $servicosProblematicos) {
          Write-Host "Reiniciando $($servico.DisplayName)..." -ForegroundColor
Yellow
          try {
            Restart-Service -Name $servico.Nome -Force -ErrorAction Stop
            Write-Host "  ✓ Reiniciado com sucesso" -ForegroundColor Green
          }
          catch {
            Write-Host "  ✗ Erro: $_" -ForegroundColor Red
          }
        }
      }
    } else {
```

```
    Write-Host "Nenhum serviço problemático encontrado." -ForegroundColor
Green
  }
}

# Função para backup de configuração de serviços
function Export-ServiceConfiguration {
  param(
    [string]$OutputPath = ".\ServiceBackup_$(Get-Date -Format
'yyyyMMdd_HHmmss').csv"
  )

  Write-Host "Exportando configuração de serviços..." -ForegroundColor Yellow

  $servicos = Get-CimInstance Win32_Service | Select-Object @{
    Name='Nome'
    Expression={$_.Name}
  }, @{
    Name='NomeExibicao'
    Expression={$_.DisplayName}
  }, @{
    Name='Status'
    Expression={$_.State}
  }, @{
    Name='TipoInicio'
    Expression={$_.StartMode}
  }, @{
    Name='ContaLogon'
    Expression={$_.StartName}
```

```
    }, @{
        Name='CaminhoExecutavel'
        Expression={$_.PathName}
    }

    $servicos | Export-Csv -Path $OutputPath -NoTypeInformation -Encoding UTF8

    Write-Host "✓ Configuração exportada para: $OutputPath" -ForegroundColor
Green
    Write-Host "Total de serviços: $($servicos.Count)" -ForegroundColor Cyan
}

# Função para criar novo serviço
function New-WindowsService {
    param(
        [Parameter(Mandatory=$true)]
        [string]$ServiceName,

        [Parameter(Mandatory=$true)]
        [string]$DisplayName,

        [Parameter(Mandatory=$true)]
        [ValidateScript({Test-Path $_})]
        [string]$BinaryPath,

        [ValidateSet("Automatic", "Manual", "Disabled")]
        [string]$StartupType = "Manual",
```

```
    [string]$Description
)

Write-Host "Criando serviço: $DisplayName" -ForegroundColor Yellow

try {
    # Criar serviço usando New-Service (requer PowerShell 6+) ou sc.exe
    if ($PSVersionTable.PSVersion.Major -ge 6) {
        $params = @{
            Name = $ServiceName
            DisplayName = $DisplayName
            BinaryPathName = $BinaryPath
            StartupType = $StartupType
        }

        if ($Description) {
            $params.Description = $Description
        }

        New-Service @params
    } else {
        # Usar sc.exe para versões antigas
        $startType = switch ($StartupType) {
            "Automatic" { "auto" }
            "Manual" { "demand" }
            "Disabled" { "disabled" }
        }
```

```
        $result = & sc.exe create $ServiceName binPath= $BinaryPath DisplayName=
$DisplayName start= $startType


        if ($Description) {
            & sc.exe description $ServiceName $Description | Out-Null
        }
    }

    Write-Host "✓ Serviço criado com sucesso!" -ForegroundColor Green

    # Exibir informações
    $novoServico = Get-Service -Name $ServiceName
    $novoServico | Format-List Name, DisplayName, Status, StartType
  }
  catch {
    Write-Host "✗ Erro ao criar serviço: $_" -ForegroundColor Red
  }
}

# Menu principal
function Show-ServiceMenu {
  do {
    Clear-Host
    Write-Host
"╔═══════════════════════════════════════════════════════════╗" -
ForegroundColor Cyan
    Write-Host "║    GERENCIADOR DE SERVIÇOS DO WINDOWS     ║" -
ForegroundColor Cyan
```

```
    Write-Host
"╚═══════════════════════════════════════════════════════════════════════════╝" -
ForegroundColor Cyan
    Write-Host ""
    Write-Host "1. Verificar Serviços Críticos" -ForegroundColor White
    Write-Host "2. Listar Serviços em Execução" -ForegroundColor White
    Write-Host "3. Listar Serviços Parados" -ForegroundColor White
    Write-Host "4. Reiniciar Serviços Problemáticos" -ForegroundColor White
    Write-Host "5. Exportar Configuração de Serviços" -ForegroundColor White
    Write-Host "6. Buscar Serviço" -ForegroundColor White
    Write-Host "7. Gerenciar Serviço Específico" -ForegroundColor White
    Write-Host "8. Sair" -ForegroundColor White
    Write-Host ""

    $opcao = Read-Host "Escolha uma opção"

    switch ($opcao) {
      "1" {
        $criticos = Get-CriticalServices
        $criticos | Format-Table -AutoSize
        Read-Host "`nPressione Enter para continuar"
      }
      "2" {
        Get-Service | Where-Object Status -eq "Running" |
          Sort-Object DisplayName |
          Format-Table Name, DisplayName, Status -AutoSize
        Read-Host "`nPressione Enter para continuar"
      }
      "3" {
```

```
            Get-Service | Where-Object Status -eq "Stopped" |
                Sort-Object DisplayName |
                Format-Table Name, DisplayName, StartType -AutoSize
            Read-Host "`nPressione Enter para continuar"
        }
        "4" {
            Restart-ProblematicServices
            Read-Host "`nPressione Enter para continuar"
        }
        "5" {
            Export-ServiceConfiguration
            Read-Host "`nPressione Enter para continuar"
        }
        "6" {
            $busca = Read-Host "Digite o nome ou parte do nome do serviço"
            Get-Service | Where-Object {
                $_.Name -like "*$busca*" -or $_.DisplayName -like "*$busca*"
            } | Format-Table Name, DisplayName, Status, StartType -AutoSize
            Read-Host "`nPressione Enter para continuar"
        }
        "7" {
            $nomeServico = Read-Host "Digite o nome do serviço"
            $servico = Get-Service -Name $nomeServico -ErrorAction SilentlyContinue

            if ($servico) {
                Write-Host "`nServiço: $($servico.DisplayName)" -ForegroundColor
Cyan
                Write-Host "Status: $($servico.Status)" -ForegroundColor
$(if($servico.Status -eq 'Running'){'Green'}else{'Red'})
```

```
            Write-Host "`n1. Iniciar"
            Write-Host "2. Parar"
            Write-Host "3. Reiniciar"
            Write-Host "4. Voltar"

            $acao = Read-Host "`nEscolha uma ação"

            switch ($acao) {
                "1" { Start-Service -Name $nomeServico; Write-Host "Serviço iniciado" -ForegroundColor Green }
                "2" { Stop-Service -Name $nomeServico; Write-Host "Serviço parado" -ForegroundColor Yellow }
                "3" { Restart-Service -Name $nomeServico; Write-Host "Serviço reiniciado" -ForegroundColor Green }
            }
        } else {
            Write-Host "Serviço não encontrado!" -ForegroundColor Red
        }
        Read-Host "`nPressione Enter para continuar"
    }
  }
} while ($opcao -ne "8")
}

# Executar menu
Show-ServiceMenu
```

5.2 Acesso Remoto e Gerenciamento de Múltiplos Computadores

5.2.1 PowerShell Remoting - Fundamentos

O que é PowerShell Remoting?

PowerShell Remoting permite executar comandos e scripts em computadores remotos através do protocolo **WS-Management (WinRM)**.

Características:

- Baseado em WS-Management (porta 5985 HTTP, 5986 HTTPS)
- Suporta autenticação Kerberos, NTLM, CredSSP
- Permite execução de comandos em múltiplos computadores simultaneamente
- Suporta sessões persistentes e interativas

Habilitar PowerShell Remoting

 # No computador de destino (servidor/alvo)

 # Executar como Administrador

Enable-PSRemoting -Force

 # Configuração automática:

 # - Inicia serviço WinRM

 # - Configura tipo de inicialização como Automático

 # - Cria listener para aceitar requisições

 # - Cria regras de firewall

 # Verificar configuração

Test-WSMan

 # Verificar listeners

Get-WSManInstance -ResourceURI winrm/config/listener -Enumerate

 # Desabilitar (se necessário)

Disable-PSRemoting -Force

Configurar Clientes Confiáveis (Workgroup)

Em ambientes sem domínio Active Directory:

 # No computador cliente

 # Executar como Administrador

 # Adicionar hosts confiáveis (TrustedHosts)

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "192.168.1.100,Server01" -Force

 # Adicionar todos (não recomendado em produção)

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "*" -Force

 # Adicionar mantendo valores existentes

$current = (Get-Item WSMan:\localhost\Client\TrustedHosts).Value

Set-Item WSMan:\localhost\Client\TrustedHosts -Value "$current,NovoServidor" -Force

 # Verificar hosts confiáveis

Get-Item WSMan:\localhost\Client\TrustedHosts

 # Limpar lista

Clear-Item WSMan:\localhost\Client\TrustedHosts -Force

Configurar Firewall

 # Regras criadas automaticamente pelo Enable-PSRemoting:

 # - Windows Remote Management (HTTP-In)

 # - Windows Remote Management - Compatibility Mode (HTTP-In)

 # Verificar regras de firewall

Get-NetFirewallRule -Name "WINRM-HTTP-In-TCP*" | Select-Object Name, Enabled, Direction

```
# Criar regra manualmente (se necessário)
New-NetFirewallRule -Name "PSRemoting-In" `
  -DisplayName "PowerShell Remoting" `
  -Protocol TCP `
  -LocalPort 5985 `
  -Action Allow `
  -Direction Inbound `
  -Enabled True

# Testar conectividade
Test-NetConnection -ComputerName "Server01" -Port 5985
```

5.2.2 Comandos Remotos

Invoke-Command - Executar Comandos Remotos

Características:

- Executa comandos em um ou mais computadores remotos
- Retorna objetos desserializados
- Suporta execução paralela

```
# Sintaxe básica
Invoke-Command -ComputerName NomeComputador -ScriptBlock { comando }

# Exemplo 1: Comando simples
Invoke-Command -ComputerName "Server01" -ScriptBlock {
  Get-Process | Select-Object -First 5
}

# Exemplo 2: Com credenciais
$cred = Get-Credential
Invoke-Command -ComputerName "Server01" -Credential $cred -ScriptBlock {
```

```
    Get-Service
}

# Exemplo 3: Múltiplos computadores
$computadores = "Server01", "Server02", "Server03"
Invoke-Command -ComputerName $computadores -ScriptBlock {
    Get-ComputerInfo | Select-Object CsName, OsName, OsVersion
}

# Exemplo 4: Usando variáveis locais
$serviceName = "wuauserv"
Invoke-Command -ComputerName "Server01" -ScriptBlock {
    Get-Service -Name $using:serviceName
}

# Exemplo 5: Executar script remoto
Invoke-Command -ComputerName "Server01" -FilePath
"C:\Scripts\MeuScript.ps1"

# Exemplo 6: Passar argumentos
Invoke-Command -ComputerName "Server01" -FilePath "C:\Scripts\Script.ps1" -
ArgumentList "arg1", "arg2"

# Exemplo 7: Execução assíncrona (job)
$job = Invoke-Command -ComputerName "Server01" -ScriptBlock {
    Get-EventLog -LogName System -Newest 1000
} -AsJob

# Aguardar job
```

```
Wait-Job $job
Receive-Job $job

# Exemplo 8: Com throttle (limitar execuções paralelas)
Invoke-Command -ComputerName $computadores -ScriptBlock {
    Get-Process
} -ThrottleLimit 10

# Exemplo 9: Salvar saída em variável
$resultados = Invoke-Command -ComputerName "Server01" -ScriptBlock {
    Get-WinEvent -LogName Application -MaxEvents 100
}

$resultados | Where-Object LevelDisplayName -eq "Error"

# Exemplo 10: Com timeout
Invoke-Command -ComputerName "Server01" -ScriptBlock {
    Start-Sleep -Seconds 60
} -SessionOption (New-PSSessionOption -IdleTimeout 30000)
```

Enter-PSSession e Exit-PSSession - Sessão Interativa

```
# Iniciar sessão interativa
Enter-PSSession -ComputerName "Server01"

# Com credenciais
$cred = Get-Credential
Enter-PSSession -ComputerName "Server01" -Credential $cred

# Prompt mudará para: [Server01]: PS C:\>
```

```
# Todos os comandos são executados no servidor remoto


# Comandos de exemplo dentro da sessão:
Get-Process
Get-Service
cd C:\Logs
Get-ChildItem


# Sair da sessão
Exit-PSSession
```

5.2.3 Sessões Persistentes (PSSessions)

PSSessions são sessões persistentes que mantêm estado entre comandos.

Vantagens:

- Melhor performance para múltiplos comandos
- Mantém variáveis e estado
- Reutilizável
- Suporta desconexão e reconexão

```
# Criar sessão
$sessao = New-PSSession -ComputerName "Server01"


# Ver informações da sessão
$sessao | Format-List *


# Usar sessão com Invoke-Command
Invoke-Command -Session $sessao -ScriptBlock {
  $variavel = "Dados persistem na sessão"
  Get-Date
}
```

```
# Reutilizar sessão (variável ainda existe)
Invoke-Command -Session $sessao -ScriptBlock {
  Write-Output $variavel  # "Dados persistem na sessão"
}

# Múltiplas sessões
$servidores = "Server01", "Server02", "Server03"
$sessoes = New-PSSession -ComputerName $servidores

# Executar em todas as sessões
Invoke-Command -Session $sessoes -ScriptBlock {
  Get-Process | Measure-Object -Property WorkingSet -Sum
}

# Fechar sessão específica
Remove-PSSession -Session $sessao

# Fechar todas as sessões
Get-PSSession | Remove-PSSession

# Sessões desconectadas (útil para tarefas longas)
$sessao = New-PSSession -ComputerName "Server01"

# Executar comando e desconectar
Invoke-Command -Session $sessao -ScriptBlock {
  Start-Sleep -Seconds 300
} -InDisconnectedSession
```

```
# Listar sessões desconectadas
Get-PSSession -ComputerName "Server01"

# Reconectar
$sessao = Get-PSSession -ComputerName "Server01" -State Disconnected
Connect-PSSession -Session $sessao

# Receber resultados
Receive-PSSession -Session $sessao
```

Opções de Sessão

```
# Criar opções customizadas
$opcoes = New-PSSessionOption `
  -IdleTimeout 7200000 `          # 2 horas
  -OperationTimeout 3600000 `     # 1 hora
  -MaxConnectionRetryCount 5 `
  -NoMachineProfile `
  -Culture "pt-BR" `
  -UICulture "pt-BR"

# Usar opções
$sessao = New-PSSession -ComputerName "Server01" -SessionOption $opcoes

# Configuração padrão de sessão
$PSDefaultParameterValues = @{
  'New-PSSession:SessionOption' = $opcoes
}
```

5.2.4 Gerenciamento de Múltiplos Computadores

Exemplo 1: Coletar Informações de Múltiplos Servidores

```
# Lista de servidores
$servidores = @(
  "Server01",
  "Server02",
  "Server03",
  "Server04",
  "Server05"
)

# Criar sessões
Write-Host "Conectando aos servidores..." -ForegroundColor Yellow
$sessoes = New-PSSession -ComputerName $servidores -ErrorAction SilentlyContinue

# Verificar conexões bem-sucedidas
$conectados = $sessoes | Select-Object -ExpandProperty ComputerName
$falhas = Compare-Object $servidores $conectados |
  Where-Object SideIndicator -eq '<=' |
  Select-Object -ExpandProperty InputObject

if ($falhas) {
  Write-Host "Falha ao conectar:" -ForegroundColor Red
  $falhas | ForEach-Object { Write-Host "  - $_" -ForegroundColor Red }
}

# Coletar informações
Write-Host "`nColetando informações..." -ForegroundColor Yellow
```

```
$informacoes = Invoke-Command -Session $sessoes -ScriptBlock {
    $os = Get-CimInstance Win32_OperatingSystem
    $cs = Get-CimInstance Win32_ComputerSystem
    $cpu = Get-CimInstance Win32_Processor
    $discos = Get-CimInstance Win32_LogicalDisk -Filter "DriveType=3"

    [PSCustomObject]@{
        Computador = $env:COMPUTERNAME
        SO = $os.Caption
        Versao = $os.Version
        UltimoReboot = $os.LastBootUpTime
        Processador = $cpu.Name
        Nucleos = $cpu.NumberOfCores
        MemoriaGB = [math]::Round($cs.TotalPhysicalMemory / 1GB, 2)
        Discos = $discos | ForEach-Object {
            [PSCustomObject]@{
                Letra = $_.DeviceID
                TamanhoGB = [math]::Round($_.Size / 1GB, 2)
                LivreGB = [math]::Round($_.FreeSpace / 1GB, 2)
                UsoPct = [math]::Round((($_.Size - $_.FreeSpace) / $_.Size) * 100, 2)
            }
        }
    }
}

# Exibir resultados
$informacoes | ForEach-Object {
```

```
  Write-Host "`n=== $($_.Computador) ===" -ForegroundColor Cyan
  Write-Host "SO: $($_.SO)" -ForegroundColor White
  Write-Host "Versão: $($_.Versao)" -ForegroundColor White
  Write-Host "Último Reboot: $($_.UltimoReboot)" -ForegroundColor White
  Write-Host "CPU: $($_.Processador) ($($_.Nucleos) núcleos)" -ForegroundColor White
  Write-Host "Memória: $($_.MemoriaGB) GB" -ForegroundColor White
  Write-Host "Discos:" -ForegroundColor White
  $_.Discos | ForEach-Object {
    $cor = if ($_.UsoPct -gt 80) { 'Red' } elseif ($_.UsoPct -gt 70) { 'Yellow' } else { 'Green' }
    Write-Host "  $($_.Letra) $($_.TamanhoGB) GB - Livre: $($_.LivreGB) GB ($($_.UsoPct)% usado)" -ForegroundColor $cor
  }
}

# Exportar para CSV
$relatorio = foreach ($info in $informacoes) {
  foreach ($disco in $info.Discos) {
    [PSCustomObject]@{
      Computador = $info.Computador
      SO = $info.SO
      Versao = $info.Versao
      UltimoReboot = $info.UltimoReboot
      CPU = $info.Processador
      Nucleos = $info.Nucleos
      MemoriaGB = $info.MemoriaGB
      Disco = $disco.Letra
      DiscoTamanhoGB = $disco.TamanhoGB
```

```
        DiscoLivreGB = $disco.LivreGB
        DiscoUsoPct = $disco.UsoPct
      }
    }
}

$relatorio | Export-Csv "Inventario_Servidores_$(Get-Date -Format
'yyyyMMdd_HHmmss').csv" -NoTypeInformation

# Fechar sessões
Remove-PSSession -Session $sessoes

Write-Host "`n✓ Coleta concluída!" -ForegroundColor Green
```

Exemplo 2: Executar Manutenção em Múltiplos Servidores

```
# Arquivo: ManutencaoServidores.ps1

<#
.SYNOPSIS
   Executa tarefas de manutenção em múltiplos servidores
#>

param(
   [Parameter(Mandatory=$true)]
   [string[]]$Servidores,

   [switch]$LimparTemp,
   [switch]$AtualizarWindows,
   [switch]$ReiniciarServicos,
```

```
    [switch]$ColetarLogs,

    [string]$LogPath = ".\MaintenanceLogs"
)

# Criar diretório de logs
if (-not (Test-Path $LogPath)) {
    New-Item -Path $LogPath -ItemType Directory | Out-Null
}

$logFile = Join-Path $LogPath "Manutencao_$(Get-Date -Format
'yyyyMMdd_HHmmss').log"

function Write-MaintenanceLog {
    param(
        [string]$Message,
        [string]$Level = "INFO"
    )

    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    $logEntry = "[$timestamp] [$Level] $Message"

    $color = switch ($Level) {
        "INFO" { "White" }
        "SUCCESS" { "Green" }
        "WARNING" { "Yellow" }
        "ERROR" { "Red" }
        default { "Gray" }
```

```
    }

    Write-Host $logEntry -ForegroundColor $color
    $logEntry | Out-File -FilePath $logFile -Append -Encoding UTF8
}

Write-MaintenanceLog "===== INÍCIO DA MANUTENÇÃO ====="
Write-MaintenanceLog "Servidores alvo: $($Servidores -join ', ')"

# Criar sessões
Write-MaintenanceLog "Estabelecendo conexões..." -Level "INFO"
$sessoes = @()

foreach ($servidor in $Servidores) {
    try {
        $sessao = New-PSSession -ComputerName $servidor -ErrorAction Stop
        $sessoes += $sessao
        Write-MaintenanceLog "✓ Conectado: $servidor" -Level "SUCCESS"
    }
    catch {
        Write-MaintenanceLog "✗ Falha ao conectar: $servidor - $_" -Level "ERROR"
    }
}

if ($sessoes.Count -eq 0) {
    Write-MaintenanceLog "Nenhuma conexão estabelecida. Encerrando." -Level
"ERROR"
    exit 1
```

```
}

# Tarefa 1: Limpar arquivos temporários
if ($LimparTemp) {
  Write-MaintenanceLog "`n--- LIMPEZA DE ARQUIVOS TEMPORÁRIOS ---" -Level "INFO"

  $resultados = Invoke-Command -Session $sessoes -ScriptBlock {
    $caminhos = @(
      "$env:TEMP",
      "C:\Windows\Temp",
      "C:\Windows\Prefetch"
    )

    $totalRemovido = 0
    $erros = @()

    foreach ($caminho in $caminhos) {
      if (Test-Path $caminho) {
        try {
          $arquivos = Get-ChildItem -Path $caminho -Recurse -File -ErrorAction SilentlyContinue |
                      Where-Object LastWriteTime -lt (Get-Date).AddDays(-7)

          $tamanho = ($arquivos | Measure-Object -Property Length -Sum).Sum
          $totalRemovido += $tamanho

          $arquivos | Remove-Item -Force -ErrorAction SilentlyContinue
        }
```

```
            catch {
                $erros += "Erro em $caminho : $_"
            }
        }
    }

    [PSCustomObject]@{
        Computador = $env:COMPUTERNAME
        RemovidoMB = [math]::Round($totalRemovido / 1MB, 2)
        Erros = $erros
    }
  }

  foreach ($resultado in $resultados) {
    Write-MaintenanceLog "$($resultado.Computador): Removidos
$($resultado.RemovidoMB) MB" -Level "SUCCESS"
    if ($resultado.Erros) {
        foreach ($erro in $resultado.Erros) {
            Write-MaintenanceLog "  $erro" -Level "WARNING"
        }
    }
  }
}

# Tarefa 2: Reiniciar serviços
if ($ReiniciarServicos) {
  Write-MaintenanceLog "`n--- REINÍCIO DE SERVIÇOS ---" -Level "INFO"
```

```
$servicosReiniciar = @("wuauserv", "BITS")

$resultados = Invoke-Command -Session $sessoes -ScriptBlock {
  param($Servicos)

  $resultados = @()
  foreach ($nomeServico in $Servicos) {
    try {
      $servico = Get-Service -Name $nomeServico -ErrorAction Stop
      if ($servico.Status -eq 'Running') {
        Restart-Service -Name $nomeServico -Force -ErrorAction Stop
        $resultados += "$nomeServico reiniciado com sucesso"
      } else {
        $resultados += "$nomeServico não estava em execução"
      }
    }
    catch {
      $resultados += "Erro ao reiniciar $nomeServico : $_"
    }
  }

  [PSCustomObject]@{
    Computador = $env:COMPUTERNAME
    Resultados = $resultados
  }
} -ArgumentList (,$servicosReiniciar)

foreach ($resultado in $resultados) {
```

```
        Write-MaintenanceLog "$($resultado.Computador):" -Level "INFO"
        foreach ($msg in $resultado.Resultados) {
            Write-MaintenanceLog "  $msg" -Level "SUCCESS"
        }
    }
}

# Tarefa 3: Coletar logs
if ($ColetarLogs) {
    Write-MaintenanceLog "`n--- COLETA DE LOGS ---" -Level "INFO"

    $logsPath = Join-Path $LogPath "ServerLogs_$(Get-Date -Format 'yyyyMMdd_HHmmss')"
    New-Item -Path $logsPath -ItemType Directory | Out-Null

    foreach ($sessao in $sessoes) {
        $servidor = $sessao.ComputerName
        Write-MaintenanceLog "Coletando logs de $servidor..." -Level "INFO"

        try {
            $eventos = Invoke-Command -Session $sessao -ScriptBlock {
                Get-WinEvent -LogName System -MaxEvents 100 |
                    Where-Object LevelDisplayName -in "Error", "Warning" |
                    Select-Object TimeCreated, LevelDisplayName, ProviderName, Message
            }

            $arquivoLog = Join-Path $logsPath "$servidor`_EventLog.csv"
            $eventos | Export-Csv -Path $arquivoLog -NoTypeInformation
```

```
        Write-MaintenanceLog "  ✓ Logs salvos: $arquivoLog" -Level "SUCCESS"
      }
      catch {
        Write-MaintenanceLog "  ✗ Erro ao coletar logs: $_" -Level "ERROR"
      }
    }
}

# Relatório final
Write-MaintenanceLog "`n--- RELATÓRIO FINAL ---" -Level "INFO"

$statusFinal = Invoke-Command -Session $sessoes -ScriptBlock {
    $os = Get-CimInstance Win32_OperatingSystem

    [PSCustomObject]@{
        Computador = $env:COMPUTERNAME
        Uptime = (Get-Date) - $os.LastBootUpTime
        MemoriaLivrePct = [math]::Round(($os.FreePhysicalMemory / $os.TotalVisibleMemorySize) * 100, 2)
        ProcessosAtivos = (Get-Process).Count
    }
}

foreach ($status in $statusFinal) {
    Write-MaintenanceLog "$($status.Computador):" -Level "INFO"
    Write-MaintenanceLog "  Uptime: $($status.Uptime.Days) dias, $($status.Uptime.Hours) horas" -Level "INFO"
```

```
    Write-MaintenanceLog "  Memória Livre: $($status.MemoriaLivrePct)%" -Level "INFO"
    Write-MaintenanceLog "  Processos Ativos: $($status.ProcessosAtivos)" -Level "INFO"
}

# Fechar sessões
Remove-PSSession -Session $sessoes
Write-MaintenanceLog "`n===== MANUTENÇÃO CONCLUÍDA =====" -Level "SUCCESS"
Write-MaintenanceLog "Log completo salvo em: $logFile" -Level "INFO"
```

Uso:

```
# Executar todas as tarefas
.\ManutencaoServidores.ps1 -Servidores "Server01", "Server02" -LimparTemp -ReiniciarServicos -ColetarLogs

# Apenas limpeza
.\ManutencaoServidores.ps1 -Servidores "Server01", "Server02", "Server03" -LimparTemp

# Coletar logs de todos os servidores
.\ManutencaoServidores.ps1 -Servidores (Get-Content .\servidores.txt) -ColetarLogs
```

5.2.5 Gerenciamento Remoto com CIM/WMI

CIM (Common Information Model) é o sucessor do WMI e funciona nativamente com PowerShell Remoting.

```
# Criar sessão CIM
$sessaoCim = New-CimSession -ComputerName "Server01"

# Usar sessão CIM
```

```
Get-CimInstance -ClassName Win32_OperatingSystem -CimSession $sessaoCim


# Múltiplas sessões CIM

$sessoesCim = New-CimSession -ComputerName "Server01", "Server02",
"Server03"


# Query em múltiplos servidores

Get-CimInstance -ClassName Win32_Service -Filter "State='Running'" -
CimSession $sessoesCim |

  Select-Object PSComputerName, Name, DisplayName, State


# Invocar método CIM

$servico = Get-CimInstance -ClassName Win32_Service -Filter
"Name='wuauserv'" -CimSession $sessaoCim

Invoke-CimMethod -InputObject $servico -MethodName StopService


# Fechar sessões

Remove-CimSession -CimSession $sessoesCim
```

5.2.6 Tratamento de Erros em Operações Remotas

```
# Capturar erros de conexão

$servidores = "Server01", "ServerInexistente", "Server02"

$sessoes = @()


foreach ($servidor in $servidores) {

  try {

    $sessao = New-PSSession -ComputerName $servidor -ErrorAction Stop

    $sessoes += $sessao

    Write-Host "✓ Conectado: $servidor" -ForegroundColor Green

  }
```

```
  catch {
    Write-Host "✗ Falha: $servidor - $($_.Exception.Message)" -ForegroundColor Red
  }
}

# Tratamento durante Invoke-Command
$resultados = Invoke-Command -ComputerName $servidores -ScriptBlock {
  try {
    Get-Service -Name "ServicoInexistente" -ErrorAction Stop
  }
  catch {
    [PSCustomObject]@{
      Computador = $env:COMPUTERNAME
      Erro = $_.Exception.Message
    }
  }
} -ErrorAction SilentlyContinue -ErrorVariable errosRemotOs

# Analisar erros
if ($errosRemotos) {
  Write-Host "Erros encontrados:" -ForegroundColor Red
  $errosRemotos | ForEach-Object {
    Write-Host "  $($_.TargetObject): $($_.Exception.Message)" -ForegroundColor Red
  }

}
```

6. MELHORES PRÁTICAS E SEGURANÇA

6.1 Gestão de Credenciais e Segurança em Scripts

6.1.1 Fundamentos de Segurança no PowerShell

Por Que a Segurança é Crítica?

Scripts PowerShell frequentemente:

- Acessam recursos críticos do sistema
- Conectam a servidores e bancos de dados
- Manipulam dados sensíveis
- Executam com privilégios elevados
- São compartilhados entre equipes

Princípios fundamentais:

1. **Least Privilege**: Executar com privilégios mínimos necessários
2. **Defense in Depth**: Múltiplas camadas de segurança
3. **Never Trust User Input**: Validar todas as entradas
4. **Secure by Default**: Configurações seguras por padrão
5. **Audit and Monitor**: Registrar e monitorar atividades

6.1.2 Gestão de Credenciais

❌ **Práticas INSEGURAS (Nunca Fazer)**

```
# ❌ NUNCA: Hardcode de senha em texto plano
$username = "admin"
$password = "SenhaSecreta123"

# ❌ NUNCA: Senha em string
$senha = "SenhaSecreta123"
$securePassword = ConvertTo-SecureString $senha -AsPlainText -Force

# ❌ NUNCA: Credenciais em comentários
# Usuário: admin
# Senha: MinhaSenha123
```

 # ❌ NUNCA: Variáveis de ambiente para senhas

$env:DB_PASSWORD = "SenhaDoDb123"

 # ❌ NUNCA: Parâmetros de linha de comando com senha

.\script.ps1 -Password "MinhaSenha"

✅ **Get-Credential - Solicitação Interativa**

 # Método seguro: solicitar credenciais em tempo de execução

$cred = Get-Credential

 # Com nome de usuário pré-preenchido

$cred = Get-Credential -UserName "DOMINIO\Usuario"

 # Com mensagem customizada

$cred = Get-Credential -Message "Digite suas credenciais para acessar o servidor SQL"

 # Usar credencial

Invoke-Command -ComputerName "Server01" -Credential $cred -ScriptBlock {

  Get-Process

}

 # Extrair componentes (quando absolutamente necessário)

$username = $cred.UserName

$password = $cred.GetNetworkCredential().Password *# ⚠️ Expõe senha em texto plano*

✅ **SecureString - Armazenamento Seguro em Memória**

 # Criar SecureString

```
$securePassword = ConvertTo-SecureString "MinhaSenh@" -AsPlainText -Force

# Criar PSCredential com SecureString
$username = "DOMINIO\Usuario"
$cred = New-Object System.Management.Automation.PSCredential($username, $securePassword)

# Converter SecureString para texto plano (quando necessário)
$BSTR = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($securePassword)
$plainPassword = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($BSTR)
[System.Runtime.InteropServices.Marshal]::ZeroFreeBSTR($BSTR)  # Limpar memória

# Verificar se SecureString está vazia
if ($securePassword.Length -eq 0) {
  Write-Host "Senha vazia"
}
```

✅ **Armazenamento Persistente de Credenciais**

Método 1: Export/Import-Clixml (Vinculado ao Usuário e Máquina)

```
# Salvar credencial
$cred = Get-Credential
$cred | Export-Clixml -Path "C:\Credentials\mycred.xml"

# Carregar credencial
$cred = Import-Clixml -Path "C:\Credentials\mycred.xml"
```

 # ⚠️ IMPORTANTE: O arquivo só pode ser descriptografado:

 # - Pelo mesmo usuário

 # - Na mesma máquina

 # - Com a mesma instalação do Windows

 # Exemplo prático: Script que usa credencial salva

$credPath = "C:\Credentials\sqlcred.xml"

if (-not (Test-Path $credPath)) {

  Write-Host "Primeira execução - configure suas credenciais:" -ForegroundColor Yellow

  $cred = Get-Credential -Message "Credenciais SQL Server"

  $cred | Export-Clixml -Path $credPath

} else {

  $cred = Import-Clixml -Path $credPath

}

 # Usar credencial

Invoke-Sqlcmd -ServerInstance "SQLServer" -Credential $cred -Query "SELECT @@VERSION"

Método 2: ConvertFrom-SecureString / ConvertTo-SecureString (com Chave)

 # Gerar chave AES (256-bit)

$key = New-Object Byte[] 32

[Security.Cryptography.RNGCryptoServiceProvider]::Create().GetBytes($key)

 # Salvar chave em arquivo seguro

$key | Out-File "C:\Secure\aes.key" -Encoding Byte

 # Criptografar senha com chave

```
$securePassword = Read-Host "Digite a senha" -AsSecureString
$encryptedPassword = $securePassword | ConvertFrom-SecureString -Key $key

# Salvar senha criptografada
$encryptedPassword | Out-File "C:\Secure\password.txt"

# ===== Em outro script ou execução =====

# Carregar chave
$key = Get-Content "C:\Secure\aes.key" -Encoding Byte

# Carregar e descriptografar senha
$encryptedPassword = Get-Content "C:\Secure\password.txt"
$securePassword = $encryptedPassword | ConvertTo-SecureString -Key $key

# Criar credencial
$username = "DOMINIO\Usuario"
$cred = New-Object System.Management.Automation.PSCredential($username, $securePassword)

# ⚠️ IMPORTANTE: Proteja o arquivo de chave com ACLs adequadas!
```

Proteger arquivo de chave com ACLs:

```
# Remover herança e permissões existentes
$acl = Get-Acl "C:\Secure\aes.key"
$acl.SetAccessRuleProtection($true, $false)

# Adicionar permissão apenas para o usuário atual
$rule = New-Object System.Security.AccessControl.FileSystemAccessRule(
```

```
    $env:USERNAME,
    "FullControl",
    "Allow"
)
$acl.AddAccessRule($rule)

# Aplicar ACL
Set-Acl "C:\Secure\aes.key" -AclObject $acl

# Verificar permissões
Get-Acl "C:\Secure\aes.key" | Format-List
```

✅ **Windows Credential Manager (CredentialManager Module)**

```
# Instalar módulo (se não estiver instalado)
Install-Module -Name CredentialManager -Force

# Salvar credencial no Credential Manager
New-StoredCredential -Target "MeuServidor" -UserName "DOMINIO\Usuario" -Password "MinhaSenh@" -Persist LocalMachine

# Recuperar credencial
$cred = Get-StoredCredential -Target "MeuServidor"

# Listar credenciais armazenadas
Get-StoredCredential

# Remover credencial
Remove-StoredCredential -Target "MeuServidor"
```

```
# Exemplo prático
function Get-SafeCredential {
  param(
    [string]$Target,
    [string]$Message = "Digite suas credenciais"
  )

  try {
    # Tentar obter do Credential Manager
    $cred = Get-StoredCredential -Target $Target -ErrorAction Stop
    if ($cred) {
      Write-Host "Credencial recuperada do Credential Manager" -
ForegroundColor Green
      return $cred
    }
  }
  catch {
    # Se não existir, solicitar e salvar
    Write-Host "Credencial não encontrada. Configurando..." -ForegroundColor
Yellow
    $cred = Get-Credential -Message $Message

    if ($cred) {
      New-StoredCredential -Target $Target `
        -UserName $cred.UserName `
        -Password $cred.GetNetworkCredential().Password `
        -Persist LocalMachine
```

```
        Write-Host "Credencial salva no Credential Manager" -ForegroundColor
Green
        return $cred
    }
  }

  return $null
}

# Usar função
$cred = Get-SafeCredential -Target "SQLServer" -Message "Credenciais SQL"
```

✅ **Azure Key Vault (Ambientes Enterprise)**

```
# Instalar módulo Azure
Install-Module -Name Az.KeyVault -Force

# Conectar ao Azure
Connect-AzAccount

# Obter segredo do Key Vault
$secretName = "DatabasePassword"
$vaultName = "MyKeyVault"

$secret = Get-AzKeyVaultSecret -VaultName $vaultName -Name $secretName
$securePassword = $secret.SecretValue

# Criar credencial
$username = Get-AzKeyVaultSecret -VaultName $vaultName -Name
"DatabaseUser"
$cred = New-Object System.Management.Automation.PSCredential(
```

```
    $username.SecretValueText,
    $securePassword
)

# Exemplo: Função para obter credencial do Key Vault
function Get-KeyVaultCredential {
    param(
        [Parameter(Mandatory=$true)]
        [string]$VaultName,

        [Parameter(Mandatory=$true)]
        [string]$UsernameSecret,

        [Parameter(Mandatory=$true)]
        [string]$PasswordSecret
    )

    try {
        $username = (Get-AzKeyVaultSecret -VaultName $VaultName -Name
$UsernameSecret).SecretValueText
        $password = (Get-AzKeyVaultSecret -VaultName $VaultName -Name
$PasswordSecret).SecretValue

        return New-Object
System.Management.Automation.PSCredential($username, $password)
    }
    catch {
        Write-Error "Erro ao obter credencial do Key Vault: $_"
        return $null
```

```
    }
}

# Usar função
$cred = Get-KeyVaultCredential -VaultName "MyVault" -UsernameSecret "DBUser" -PasswordSecret "DBPass"
```

6.1.3 Validação e Sanitização de Entrada

Validação de Parâmetros

```
# Validações de parâmetros
param(
    # Não pode ser nulo ou vazio
    [Parameter(Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [string]$Nome,

    # Comprimento da string
    [ValidateLength(8, 50)]
    [string]$Senha,

    # Intervalo numérico
    [ValidateRange(1, 100)]
    [int]$Porcentagem,

    # Conjunto de valores permitidos
    [ValidateSet("Dev", "QA", "Prod")]
    [string]$Ambiente,

    # Padrão regex
```

```
    [ValidatePattern("^[A-Z]{2}\d{4}$")]
    [string]$Codigo,

    # Script de validação customizado
    [ValidateScript({
        Test-Path $_ -PathType Container
    })]
    [string]$Caminho,

    # Validação de contagem em arrays
    [ValidateCount(1, 10)]
    [string[]]$Servidores
)

# Exemplo: Validação de IP
param(
    [ValidatePattern("^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$")]
    [string]$EnderecoIP
)

# Exemplo: Validação de email
param(
    [ValidatePattern("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")]
    [string]$Email
)

# Exemplo: Validação de CPF
```

```
param(
  [ValidatePattern("^\d{3}\.\d{3}\.\d{3}-\d{2}$")]
  [string]$CPF
)
```

Sanitização de Entrada

```
# Função para sanitizar entrada
function ConvertTo-SafeString {
  param(
    [string]$InputString,
    [switch]$AllowSpaces
  )

  # Remover caracteres perigosos
  $safe = $InputString -replace '[^\w\s-]', ''

  if (-not $AllowSpaces) {
    $safe = $safe -replace '\s', ''
  }

  # Limitar tamanho
  if ($safe.Length -gt 100) {
    $safe = $safe.Substring(0, 100)
  }

  return $safe.Trim()
}

# Uso
```

```
$userInput = Read-Host "Digite o nome do arquivo"
$safeFilename = ConvertTo-SafeString -InputString $userInput
New-Item -Path "C:\Files\$safeFilename.txt" -ItemType File

# Sanitizar caminhos
function Test-SafePath {
  param([string]$Path)

  # Verificar caracteres inválidos
  $invalidChars = [System.IO.Path]::GetInvalidPathChars()
  foreach ($char in $invalidChars) {
    if ($Path.Contains($char)) {
      throw "Caminho contém caractere inválido: $char"
    }
  }

  # Verificar path traversal
  if ($Path -match '\.\.|~') {
    throw "Caminho contém sequência suspeita"
  }

  # Resolver para caminho absoluto
  $resolvedPath = [System.IO.Path]::GetFullPath($Path)

  # Verificar se está dentro do diretório permitido
  $allowedBase = "C:\AllowedDirectory"
  if (-not $resolvedPath.StartsWith($allowedBase)) {
    throw "Caminho fora do diretório permitido"
```

```
    }

    return $resolvedPath
}

# Sanitizar SQL (prevenção de SQL Injection)
function Invoke-SafeSqlQuery {
    param(
        [string]$Query,
        [hashtable]$Parameters
    )

    # NUNCA concatenar strings diretamente
    # ❌ ERRADO: "SELECT * FROM Users WHERE Name = '$userName'"

    # ✅ CORRETO: Usar parâmetros
    $connection = New-Object
System.Data.SqlClient.SqlConnection($connectionString)
    $command = $connection.CreateCommand()
    $command.CommandText = $Query

    foreach ($param in $Parameters.GetEnumerator()) {
        $command.Parameters.AddWithValue($param.Key, $param.Value) | Out-Null
    }

    # Executar query...
}
```

 # Uso

Invoke-SafeSqlQuery -Query "SELECT * FROM Users WHERE Name = @Name" -Parameters @{

"@Name" = $userName

}

6.1.4 Execution Policy e Script Signing

Execution Policy

 # Verificar política atual

Get-ExecutionPolicy

Get-ExecutionPolicy -List

 # Definir políticas por escopo

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

Set-ExecutionPolicy -ExecutionPolicy AllSigned -Scope LocalMachine

 # Bypass temporário (apenas para sessão)

powershell.exe -ExecutionPolicy Bypass -File script.ps1

 # Níveis de Execution Policy:

 # - Restricted: Não executa scripts (padrão Windows cliente)

 # - AllSigned: Apenas scripts assinados

 # - RemoteSigned: Scripts locais livres, remotos assinados (recomendado)

 # - Unrestricted: Todos os scripts (avisa para remotos)

 # - Bypass: Nenhuma verificação

 # - Undefined: Remove política do escopo

Assinatura Digital de Scripts

Criar certificado de assinatura de código (desenvolvimento):

 # Criar certificado self-signed (desenvolvimento apenas)

```
$cert = New-SelfSignedCertificate `
  -Type CodeSigningCert `
  -Subject "CN=PowerShell Code Signing" `
  -CertStoreLocation "Cert:\CurrentUser\My" `
  -KeyExportPolicy Exportable

# Mover para Trusted Publishers
$store = New-Object System.Security.Cryptography.X509Certificates.X509Store(
  "TrustedPublisher",
  "CurrentUser"
)
$store.Open("ReadWrite")
$store.Add($cert)
$store.Close()

# Mover para Root
$store = New-Object System.Security.Cryptography.X509Certificates.X509Store(
  "Root",
  "CurrentUser"
)
$store.Open("ReadWrite")
$store.Add($cert)
$store.Close()
```

Assinar script:

```
# Obter certificado
$cert = Get-ChildItem Cert:\CurrentUser\My -CodeSigningCert | Select-Object -First 1
```

```
# Assinar script
Set-AuthenticodeSignature -FilePath "C:\Scripts\MeuScript.ps1" -Certificate $cert

# Verificar assinatura
Get-AuthenticodeSignature -FilePath "C:\Scripts\MeuScript.ps1"

# Função para assinar todos os scripts em diretório
function Sign-AllScripts {
  param(
    [Parameter(Mandatory=$true)]
    [string]$Path,

    [Parameter(Mandatory=$true)]
    [System.Security.Cryptography.X509Certificates.X509Certificate2]$Certificate
  )

  Get-ChildItem -Path $Path -Filter "*.ps1" -Recurse | ForEach-Object {
    Write-Host "Assinando: $($_.Name)" -ForegroundColor Yellow

    try {
      Set-AuthenticodeSignature -FilePath $_.FullName -Certificate $Certificate
      Write-Host "  ✓ Assinado com sucesso" -ForegroundColor Green
    }
    catch {
      Write-Host "  ✗ Erro: $_" -ForegroundColor Red
    }
  }
}
```

 # Usar função

$cert = Get-ChildItem Cert:\CurrentUser\My -CodeSigningCert | Select-Object -First 1

Sign-AllScripts -Path "C:\Scripts" -Certificate $cert

6.1.5 Logging e Auditoria

Transcript - Registro de Sessão

 # Iniciar transcript

Start-Transcript -Path "C:\Logs\Session_$(Get-Date -Format 'yyyyMMdd_HHmmss').log"

 # Comandos executados...

Get-Process

Get-Service

 # Parar transcript

Stop-Transcript

 # Transcript automático em script

$transcriptPath = "C:\Logs\Script_$(Get-Date -Format 'yyyyMMdd_HHmmss').log"

try {

  Start-Transcript -Path $transcriptPath -ErrorAction Stop

  # Código do script...

  Write-Host "Executando operações..."

  Stop-Transcript

}

```
catch {
  if ((Get-Command Stop-Transcript -ErrorAction SilentlyContinue)) {
    Stop-Transcript
  }
  throw
}
```

Script Block Logging (Auditoria Nativa)

```
# Habilitar Script Block Logging via Group Policy ou Registry

# Via Registry (requer reinício)
$regPath = "HKLM:\SOFTWARE\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging"

if (-not (Test-Path $regPath)) {
  New-Item -Path $regPath -Force | Out-Null
}

New-ItemProperty -Path $regPath -Name "EnableScriptBlockLogging" -Value 1 -PropertyType DWORD -Force

# Visualizar logs (Event Viewer)
Get-WinEvent -LogName "Microsoft-Windows-PowerShell/Operational" |
  Where-Object Id -eq 4104 |
  Select-Object -First 10 TimeCreated, Message
```

Sistema de Logging Customizado

```
# Sistema completo de logging
enum LogLevel {
  DEBUG = 0
  INFO = 1
```

```
    WARNING = 2
    ERROR = 3
    CRITICAL = 4
}

class Logger {
    [string]$LogPath
    [string]$LogFile
    [LogLevel]$MinLevel
    [bool]$ConsoleOutput
    [bool]$FileOutput

    Logger([string]$logPath) {
        $this.LogPath = $logPath
        $this.LogFile = "Log_$(Get-Date -Format 'yyyyMMdd').log"
        $this.MinLevel = [LogLevel]::INFO
        $this.ConsoleOutput = $true
        $this.FileOutput = $true

        # Criar diretório se não existir
        if (-not (Test-Path $this.LogPath)) {
            New-Item -Path $this.LogPath -ItemType Directory -Force | Out-Null
        }
    }

    [void] Write([string]$message, [LogLevel]$level, [string]$source) {
        if ($level -lt $this.MinLevel) {
            return
```

```
    }

    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss.fff"
    $levelStr = $level.ToString().PadRight(8)
    $logEntry = "[$timestamp] [$levelStr] [$source] $message"

    # Console output
    if ($this.ConsoleOutput) {
        $color = switch ($level) {
            ([LogLevel]::DEBUG) { "Gray" }
            ([LogLevel]::INFO) { "White" }
            ([LogLevel]::WARNING) { "Yellow" }
            ([LogLevel]::ERROR) { "Red" }
            ([LogLevel]::CRITICAL) { "Magenta" }
        }
        Write-Host $logEntry -ForegroundColor $color
    }

    # File output
    if ($this.FileOutput) {
        $logFilePath = Join-Path $this.LogPath $this.LogFile
        $logEntry | Out-File -FilePath $logFilePath -Append -Encoding UTF8
    }
}

[void] Debug([string]$message, [string]$source = "SYSTEM") {
    $this.Write($message, [LogLevel]::DEBUG, $source)
}
```

```
    [void] Info([string]$message, [string]$source = "SYSTEM") {
        $this.Write($message, [LogLevel]::INFO, $source)
    }

    [void] Warning([string]$message, [string]$source = "SYSTEM") {
        $this.Write($message, [LogLevel]::WARNING, $source)
    }

    [void] Error([string]$message, [string]$source = "SYSTEM") {
        $this.Write($message, [LogLevel]::ERROR, $source)
    }

    [void] Critical([string]$message, [string]$source = "SYSTEM") {
        $this.Write($message, [LogLevel]::CRITICAL, $source)
    }

    [void] Exception([System.Management.Automation.ErrorRecord]$exception,
[string]$source = "SYSTEM") {
        $message = "Exception: $($exception.Exception.Message)`n"
        $message += "ScriptStackTrace: $($exception.ScriptStackTrace)"
        $this.Write($message, [LogLevel]::ERROR, $source)
    }
}

# Uso do Logger
$logger = [Logger]::new("C:\Logs\MeuApp")
$logger.MinLevel = [LogLevel]::DEBUG
```

```
$logger.Info("Aplicação iniciada")
$logger.Debug("Valor da variável X: 10")
$logger.Warning("Conexão lenta detectada")

try {
  Get-Item "C:\arquivo_inexistente.txt" -ErrorAction Stop
}
catch {
  $logger.Exception($_, "FILESYSTEM")
}

$logger.Critical("Sistema indisponível")
```

6.1.6 Segurança em Remoting

Configuração Segura de PSRemoting

```
# Configurar PSRemoting com HTTPS
winrm quickconfig -transport:https

# Criar certificado SSL para WinRM
$cert = New-SelfSignedCertificate -DnsName $env:COMPUTERNAME -
CertStoreLocation Cert:\LocalMachine\My

# Configurar listener HTTPS
New-Item -Path WSMan:\Localhost\Listener -Transport HTTPS -Address * -
CertificateThumbPrint $cert.Thumbprint -Force

# Verificar listeners
Get-WSManInstance -ResourceURI winrm/config/listener -Enumerate
```

```
# Configurar autenticação
Set-Item WSMan:\localhost\Service\Auth\Basic -Value $false
Set-Item WSMan:\localhost\Service\Auth\Kerberos -Value $true

# Configurar criptografia
Set-Item WSMan:\localhost\Service\AllowUnencrypted -Value $false
```

Just Enough Administration (JEA)

JEA permite delegar privilégios administrativos específicos sem dar acesso completo.

```
# Criar configuração de sessão JEA
$jeaConfigPath = "C:\JEA"
New-Item -Path $jeaConfigPath -ItemType Directory -Force

# Criar arquivo de capacidades de role (Role Capability)
$roleCapabilityPath = Join-Path $jeaConfigPath "RestartService.psrc"

@{
  GUID = [Guid]::NewGuid().ToString()
  Author = 'Admin'
  Description = 'Permite reiniciar serviços específicos'
  ModulesToImport = 'Microsoft.PowerShell.Management'
  VisibleCmdlets = @(
    @{
      Name = 'Restart-Service'
      Parameters = @{
        Name = 'Name'
        ValidateSet = 'wuauserv', 'BITS', 'Spooler'
      }
```

```
    },
    'Get-Service'
  )
  VisibleFunctions = @()
  VisibleExternalCommands = @()
} | Export-PSRoleCapabilityFile -Path $roleCapabilityPath

# Criar arquivo de configuração de sessão
$sessionConfigPath = Join-Path $jeaConfigPath "RestartServiceConfig.pssc"

@{
  SchemaVersion = '2.0.0.0'
  GUID = [Guid]::NewGuid().ToString()
  Author = 'Admin'
  Description = 'Endpoint JEA para reiniciar serviços'
  SessionType = 'RestrictedRemoteServer'
  TranscriptDirectory = 'C:\JEATranscripts'
  RunAsVirtualAccount = $true
  RoleDefinitions = @{
    'DOMINIO\ServiceOperators' = @{
      RoleCapabilityFiles = $roleCapabilityPath
    }
  }
} | Export-PSSessionConfigurationFile -Path $sessionConfigPath

# Registrar configuração
Register-PSSessionConfiguration -Name "RestartService" -Path
$sessionConfigPath -Force
```

```
# Usuários do grupo ServiceOperators podem conectar:
Enter-PSSession -ComputerName localhost -ConfigurationName RestartService

# Dentro da sessão, apenas comandos permitidos funcionam
Get-Service wuauserv
Restart-Service -Name wuauserv  # ✅ Permitido
Stop-Process -Name notepad      # ❌ Negado
```

6.1.7 Checklist de Segurança

```
# Template de script seguro
<#
.SYNOPSIS
   [Descrição do script]
.NOTES
   Autor: [Nome]
   Data: [Data]
   Versão: 1.0
   Requer: PowerShell 5.1+
   Execução: Requer privilégios elevados
#>

[CmdletBinding()]
param(
   # Parâmetros com validação adequada
   [Parameter(Mandatory=$true)]
   [ValidateNotNullOrEmpty()]
   [string]$Parametro1,
```

```
    [ValidateSet("Valor1", "Valor2")]
    [string]$Parametro2 = "Valor1"
)

# Requer versão mínima do PowerShell
#Requires -Version 5.1

# Requer execução como administrador (se necessário)
#Requires -RunAsAdministrator

# Strict mode (boas práticas)
Set-StrictMode -Version Latest
$ErrorActionPreference = "Stop"

# Logging
$logPath = "C:\Logs\MeuScript"
if (-not (Test-Path $logPath)) {
    New-Item -Path $logPath -ItemType Directory -Force | Out-Null
}

$transcriptFile = Join-Path $logPath "Transcript_$(Get-Date -Format
'yyyyMMdd_HHmmss').log"
Start-Transcript -Path $transcriptFile

try {
    # Validar ambiente
    Write-Host "Validando ambiente..." -ForegroundColor Yellow
```

```
    if (-not (Test-Path "C:\RequiredPath")) {
        throw "Diretório obrigatório não encontrado"
    }

    # Obter credenciais de forma segura
    $credPath = "C:\Secure\credentials.xml"

    if (Test-Path $credPath) {
        $cred = Import-Clixml -Path $credPath
    } else {
        $cred = Get-Credential -Message "Digite suas credenciais"
        $cred | Export-Clixml -Path $credPath
    }

    # Validar credenciais
    if (-not $cred) {
        throw "Credenciais não fornecidas"
    }

    # Código principal do script
    Write-Host "Executando operações..." -ForegroundColor Green

    # ... código aqui ...

    Write-Host "Script concluído com sucesso!" -ForegroundColor Green
}
catch {
    Write-Host "ERRO: $($_.Exception.Message)" -ForegroundColor Red
```

```
    Write-Host "Linha: $($_.InvocationInfo.ScriptLineNumber)" -ForegroundColor
Red
    Write-Host "StackTrace: $($_.ScriptStackTrace)" -ForegroundColor Red
    exit 1
}
finally {
    # Limpeza
    Stop-Transcript

    # Limpar variáveis sensíveis
    if ($cred) {
        Remove-Variable -Name cred -ErrorAction SilentlyContinue
    }
}
```

6.2 Padronização e Documentação

6.2.1 Convenções de Nomenclatura

Convenções PowerShell

```
# ✅ Cmdlets: Verbo-Substantivo (PascalCase)
Get-Process
Set-Location
New-Item

# ✅ Funções: Verbo-Substantivo (PascalCase)
function Get-UserInformation { }
function Set-ServerConfiguration { }

# ✅ Variáveis: camelCase ou PascalCase
$userName = "João"
```

```
$serverList = @()
$UserName = "João"  # Também aceitável

# ✅ Constantes: UPPER_CASE ou PascalCase
$MAX_RETRIES = 5
$ConnectionTimeout = 30

# ✅ Parâmetros: PascalCase
param(
  [string]$ComputerName,
  [int]$RetryCount
)

# ✅ Booleanos: Prefixo Is/Has/Should
$isEnabled = $true
$hasAccess = $false
$shouldContinue = $true

# ❌ Evitar abreviações não óbvias
$usrNm = "João"      # ❌ Ruim
$userName = "João"   # ✅ Bom

# ❌ Evitar underscores em funções públicas
function Get_User_Info { } # ❌ Ruim
function Get-UserInfo { }  # ✅ Bom
```

Verbos Aprovados

```
# Sempre usar verbos aprovados
```

Get-Verb | Sort-Object Verb

 # Grupos principais:

 # Common: Get, Set, New, Remove, Add, Clear, Copy, Move, etc.

 # Data: Import, Export, Convert, Select, Compare, etc.

 # Lifecycle: Start, Stop, Restart, Suspend, Resume, etc.

 # Diagnostic: Debug, Test, Trace, Measure, etc.

 # Communications: Connect, Disconnect, Read, Write, Send, Receive, etc.

 # Verificar se verbo é aprovado

Get-Verb -Verb "Fetch" *# Não é aprovado - usar Get*

Get-Verb -Verb "Get" *# ✅ Aprovado*

6.2.2 Estrutura de Scripts e Módulos

Estrutura de Script Profissional

<#

.SYNOPSIS

  Breve descrição de uma linha do que o script faz

.DESCRIPTION

  Descrição detalhada e completa do script, incluindo:

  - O que ele faz

  - Como funciona

  - Requisitos

  - Dependências

.PARAMETER ComputerName

  Nome ou endereço IP do computador alvo.

  Aceita múltiplos valores via pipeline.

.PARAMETER Credential

Credenciais para conexão remota.

Se não fornecido, usa credenciais do usuário atual.

.PARAMETER LogPath

Caminho para salvar logs.

Padrão: C:\Logs\ScriptName

.EXAMPLE

.\MeuScript.ps1 -ComputerName "Server01"

Executa o script no servidor Server01 usando credenciais atuais.

.EXAMPLE

.\MeuScript.ps1 -ComputerName "Server01" -Credential (Get-Credential) -LogPath "D:\Logs"

Executa o script com credenciais específicas e caminho de log customizado.

.EXAMPLE

Get-Content servers.txt | .\MeuScript.ps1

Processa lista de servidores via pipeline.

.INPUTS

System.String

Aceita nomes de computadores via pipeline.

.OUTPUTS

PSCustomObject

Retorna objeto com resultados da operação.

.NOTES

Arquivo: MeuScript.ps1

Autor: João Silva <joao.silva@empresa.com>

Data Criação: 15/10/2025

Última Modificação: 15/10/2025

Versão: 1.0.0

Requisitos:

- PowerShell 5.1 ou superior

- Módulo ActiveDirectory

- Permissões de administrador

Changelog:

v1.0.0 (15/10/2025) - Versão inicial

.LINK

https://docs.empresa.com/scripts/meuscript

.LINK

https://github.com/empresa/powershell-scripts

#>

#Requires -Version 5.1

```
#Requires -Modules ActiveDirectory
#Requires -RunAsAdministrator

[CmdletBinding(
  SupportsShouldProcess = $true,
  ConfirmImpact = 'Medium'
)]
param(
  [Parameter(
    Mandatory = $true,
    Position = 0,
    ValueFromPipeline = $true,
    ValueFromPipelineByPropertyName = $true,
    HelpMessage = "Nome do computador alvo"
  )]
  [ValidateNotNullOrEmpty()]
  [Alias("CN", "Server")]
  [string[]]$ComputerName,

  [Parameter(Mandatory = $false)]
  [System.Management.Automation.PSCredential]
  [System.Management.Automation.Credential()]
  $Credential = [System.Management.Automation.PSCredential]::Empty,

  [Parameter(Mandatory = $false)]
  [ValidateScript({
    if (-not (Test-Path $_)) {
      New-Item -Path $_ -ItemType Directory -Force | Out-Null
```

```
        }
        $true
    })]
    [string]$LogPath = "C:\Logs\MeuScript"
)

begin {
    # Configurações iniciais
    Set-StrictMode -Version Latest
    $ErrorActionPreference = "Stop"

    # Variáveis do script
    $script:StartTime = Get-Date
    $script:TotalProcessed = 0
    $script:SuccessCount = 0
    $script:FailureCount = 0

    # Inicializar logging
    $logFile = Join-Path $LogPath "Log_$(Get-Date -Format
'yyyyMMdd_HHmmss').log"
    Start-Transcript -Path $logFile

    Write-Verbose "Script iniciado em: $script:StartTime"
    Write-Verbose "Parâmetros: ComputerName=$ComputerName,
LogPath=$LogPath"

    # Funções internas
    function Write-Log {
        param(
```

```
        [string]$Message,
        [ValidateSet("INFO", "WARNING", "ERROR", "SUCCESS")]
        [string]$Level = "INFO"
    )

    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    $color = switch ($Level) {
        "INFO"    { "White" }
        "WARNING" { "Yellow" }
        "ERROR"   { "Red" }
        "SUCCESS" { "Green" }
    }

    $logMessage = "[$timestamp] [$Level] $Message"
    Write-Host $logMessage -ForegroundColor $color
  }

  Write-Log "===== INÍCIO DA EXECUÇÃO =====" -Level "INFO"
}

process {
  foreach ($computer in $ComputerName) {
    $script:TotalProcessed++

    Write-Log "Processando: $computer" -Level "INFO"

    # Verificar se deve processar (WhatIf)
    if ($PSCmdlet.ShouldProcess($computer, "Processar computador")) {
```

```
        try {
            # Código principal aqui

            # Simulação de processamento
            Write-Verbose "Conectando a $computer..."
            $result = Test-Connection -ComputerName $computer -Count 1 -Quiet

            if ($result) {
                Write-Log "✓ $computer acessível" -Level "SUCCESS"
                $script:SuccessCount++
            } else {
                throw "Computador não acessível"
            }
        }
        catch {
            Write-Log "✗ Erro ao processar $computer : $($_.Exception.Message)" -
Level "ERROR"
            Write-Verbose "StackTrace: $($_.ScriptStackTrace)"
            $script:FailureCount++
        }
    }
  }
}

end {
  # Finalização
  $script:EndTime = Get-Date
  $duration = $script:EndTime - $script:StartTime
```

```
    Write-Log "`n===== RESUMO DA EXECUÇÃO =====" -Level "INFO"
    Write-Log "Total processado: $script:TotalProcessed" -Level "INFO"
    Write-Log "Sucessos: $script:SuccessCount" -Level "SUCCESS"
    Write-Log "Falhas: $script:FailureCount" -Level $(if ($script:FailureCount -gt 0) { "WARNING" } else { "INFO" })
    Write-Log "Duração: $($duration.ToString('hh\:mm\:ss'))" -Level "INFO"
    Write-Log "Log salvo em: $logFile" -Level "INFO"

    Stop-Transcript

    # Retornar código de saída apropriado
    if ($script:FailureCount -gt 0) {
        exit 1
    } else {
        exit 0
    }
}
```

Estrutura de Módulo

```
ModuleName/
│
├── ModuleName.psd1 # Manifesto do módulo
├── ModuleName.psm1 # Arquivo principal do módulo
│
├── Public/ # Funções públicas (exportadas)
│   ├── Get-Something.ps1
│   ├── Set-Something.ps1
│   └── New-Something.ps1
```

```
│
├── Private/ # Funções privadas (internas)
│ ├── Helper-Function.ps1
│ └── Internal-Process.ps1
│
├── Classes/ # Classes PowerShell
│ └── MyClass.ps1
│
├── Data/ # Arquivos de dados
│ └── config.json
│
├── Tests/ # Testes Pester
│ ├── ModuleName.Tests.ps1
│ └── Integration.Tests.ps1
│
├── Docs/ # Documentação
│ ├── README.md
│ └── Examples.md
│
└── LICENSE # Licença
```

Exemplo de Manifesto (ModuleName.psd1):

```
@{
  # Informações básicas
  RootModule = 'ModuleName.psm1'
  ModuleVersion = '1.0.0'
  GUID = 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'

  # Metadados
```

```
Author = 'João Silva'
CompanyName = 'Empresa Ltda'
Copyright = '(c) 2025 Empresa Ltda. Todos os direitos reservados.'
Description = 'Módulo para gerenciamento de recursos'

# Requisitos
PowerShellVersion = '5.1'
RequiredModules = @('ActiveDirectory', 'SqlServer')

# Funções exportadas
FunctionsToExport = @(
  'Get-Something',
  'Set-Something',
  'New-Something'
)

CmdletsToExport = @()
VariablesToExport = @()
AliasesToExport = @()

# Links
PrivateData = @{
  PSData = @{
    Tags = @('Management', 'Automation')
    LicenseUri = 'https://github.com/empresa/modulename/LICENSE'
    ProjectUri = 'https://github.com/empresa/modulename'
    IconUri = ''
    ReleaseNotes = 'Versão inicial do módulo'
```

```
    }
  }
}
```

Exemplo de Módulo Principal (ModuleName.psm1):

```
# Carregar funções públicas
$publicFunctions = Get-ChildItem -Path "$PSScriptRoot\Public\*.ps1" -ErrorAction SilentlyContinue

foreach ($function in $publicFunctions) {
  try {
    . $function.FullName
    Write-Verbose "Função carregada: $($function.BaseName)"
  }
  catch {
    Write-Error "Erro ao carregar função $($function.BaseName): $_"
  }
}

# Carregar funções privadas
$privateFunctions = Get-ChildItem -Path "$PSScriptRoot\Private\*.ps1" -ErrorAction SilentlyContinue

foreach ($function in $privateFunctions) {
  try {
    . $function.FullName
  }
  catch {
    Write-Error "Erro ao carregar função privada $($function.BaseName): $_"
  }
```

```
}

# Exportar apenas funções públicas
Export-ModuleMember -Function $publicFunctions.BaseName

# Mensagem de carregamento
Write-Verbose "Módulo ModuleName carregado com sucesso"
```

6.2.3 Comentários e Documentação

Comment-Based Help

```
function Get-UserReport {
  <#
  .SYNOPSIS
    Gera relatório de usuários do Active Directory

  .DESCRIPTION
    Esta função gera um relatório detalhado de usuários do Active Directory,
    incluindo informações como:
    - Nome completo
    - Email
    - Departamento
    - Último logon
    - Status da conta

    O relatório pode ser filtrado por OU, departamento ou grupo.

  .PARAMETER OrganizationalUnit
    Distinguished Name da Unidade Organizacional.
    Se não especificado, busca em todo o domínio.
```

.PARAMETER Department

Filtra usuários por departamento.

Aceita wildcards (e ?).*

.PARAMETER IncludeDisabled

Inclui contas desabilitadas no relatório.

Por padrão, apenas contas ativas são incluídas.

.PARAMETER ExportPath

Caminho para exportar o relatório em CSV.

Se não especificado, retorna objetos no console.

.EXAMPLE

Get-UserReport

Gera relatório de todos os usuários ativos do domínio.

.EXAMPLE

Get-UserReport -Department "TI" -IncludeDisabled

Gera relatório de usuários do departamento TI, incluindo contas desabilitadas.

.EXAMPLE

Get-UserReport -OrganizationalUnit "OU=Users,DC=empresa,DC=com" -ExportPath "C:\Reports\users.csv"

Gera relatório de usuários da OU especificada e exporta para CSV.

.EXAMPLE

Get-UserReport -Department "TI" | Where-Object LastLogon -lt (Get-Date).AddDays(-30)

Busca usuários de TI que não fazem logon há mais de 30 dias.

.INPUTS

None

Esta função não aceita entrada de pipeline.

.OUTPUTS

PSCustomObject

Retorna objetos customizados com as seguintes propriedades:

- Name: Nome completo

- SamAccountName: Login

- Email: Endereço de email

- do último logon

- Enabled: Status da conta

- OU: Unidade Organizacional

.NOTES

Nome: Get-UserReport

Autor: João Silva

Versão: 1.0.0

Data: 15/10/2025

Requisitos:

```
    - Módulo ActiveDirectory
    - Permissões de leitura no AD

    Alterações:
    v1.0.0 (15/10/2025) - Versão inicial

.LINK
    Get-ADUser

.LINK
    https://docs.empresa.com/powershell/get-userreport
#>

[CmdletBinding()]
param(
    [Parameter(Mandatory=$false)]
    [string]$OrganizationalUnit,

    [Parameter(Mandatory=$false)]
    [string]$Department,

    [switch]$IncludeDisabled,

    [Parameter(Mandatory=$false)]
    [ValidateScript({Test-Path (Split-Path $_)})]
    [string]$ExportPath
)
```

```
    # Código da função...
}
```

Comentários Inline

```
function Process-Data {
    param($Data)

    # ✅ Bons comentários: Explicam o "porquê"

    # Normalizar dados antes do processamento para garantir consistência
    # Isso é necessário porque a fonte de dados pode ter formatos variados
    $normalizedData = $Data | ForEach-Object {
        $_.Trim().ToLower()
    }

    # Usar hash table para lookup O(1) ao invés de array O(n)
    # Melhora performance em grandes volumes de dados
    $lookup = @{}
    foreach ($item in $normalizedData) {
        $lookup[$item] = $true
    }

    # ❌ Comentários ruins: Repetem o código

    # Incrementa contador
    $counter++

    # Loop foreach
    foreach ($item in $items) {
```

```
    # Processa item
    Process-Item $item
  }

  # ✅ Comentários de TODO, FIXME, HACK

  # TODO: Implementar cache para melhorar performance
  # FIXME: Este código falha quando $Data está vazio
  # HACK: Workaround temporário para bug #1234 - remover após fix upstream

  # ✅ Comentários de região (para código longo)

  #region Validação de Entrada
  if (-not $Data) {
    throw "Data não pode ser nulo"
  }
  #endregion

  #region Processamento Principal
  # ... código ...
  #endregion

  #region Limpeza e Finalização
  # ... código ...
  #endregion
}
```

6.2.4 Versionamento Semântico

```
# Formato: MAJOR.MINOR.PATCH
```

```
# Exemplo: 2.3.1

# MAJOR: Mudanças incompatíveis na API
#   - Remoção de funções/parâmetros
#   - Mudança de comportamento que quebra código existente
#   - Exemplo: 1.5.2 -> 2.0.0

# MINOR: Nova funcionalidade compatível
#   - Adição de novas funções
#   - Novos parâmetros opcionais
#   - Melhorias que não quebram código existente
#   - Exemplo: 1.5.2 -> 1.6.0

# PATCH: Correções de bugs compatíveis
#   - Correção de bugs
#   - Melhorias de performance
#   - Atualizações de documentação
#   - Exemplo: 1.5.2 -> 1.5.3

# Exemplo de controle de versão em módulo
@{
  ModuleVersion = '2.3.1'

  PrivateData = @{
    PSData = @{
      ReleaseNotes = @"
v2.3.1 (15/10/2025)
- [PATCH] Corrigido bug no Get-UserReport com OUs especiais
```

- [PATCH] Melhorada performance do Export em 20%

v2.3.0 (10/10/2025)

- [MINOR] Adicionado parâmetro -IncludeGroups ao Get-UserReport

- [MINOR] Nova função Get-UserPermissions

- [PATCH] Corrigidos avisos do PSScriptAnalyzer

v2.0.0 (01/09/2025)

- [MAJOR] BREAKING: Removido parâmetro -LegacyFormat

- [MAJOR] BREAKING: Get-UserReport agora retorna [PSCustomObject] ao invés de [Hashtable]

- [MINOR] Adicionado suporte a pipeline

- [MINOR] Melhorado tratamento de erros

"@

    }

  }

}

6.2.5 Code Review Checklist

<#

CHECKLIST DE REVISÃO DE CÓDIGO

☐ *FUNCIONALIDADE*

  ☐ *O código faz o que deveria fazer?*

  ☐ *A lógica está correta?*

  ☐ *Todos os casos extremos estão cobertos?*

  ☐ *Há validação de entrada adequada?*

☐ *LEGIBILIDADE*

☐ *O código é fácil de entender?*

☐ *Nomes de variáveis e funções são descritivos?*

☐ *Convenções de nomenclatura são seguidas?*

☐ *Há comentários onde necessário (não em excesso)?*

☐ *SEGURANÇA*

☐ *Credenciais são gerenciadas de forma segura?*

☐ *Não há hardcoding de senhas ou segredos?*

☐ *Entrada do usuário é validada e sanitizada?*

☐ *Há tratamento adequado de erros?*

☐ *Logging não expõe informações sensíveis?*

☐ *PERFORMANCE*

☐ *Algoritmos são eficientes?*

☐ *Não há loops desnecessários?*

☐ *Recursos são liberados adequadamente?*

☐ *Há uso apropriado de pipeline?*

☐ *MANUTENIBILIDADE*

☐ *Código está bem estruturado?*

☐ *Funções têm responsabilidade única?*

☐ *Há duplicação de código?*

☐ *Dependências são claras?*

☐ *TESTABILIDADE*

☐ *Código é testável?*

☐ *Há testes unitários?*

☐ *Casos de erro estão testados?*

☐ *DOCUMENTAÇÃO*

☐ *Há comment-based help?*

☐ *Parâmetros estão documentados?*

☐ *Exemplos são fornecidos?*

☐ *README está atualizado?*

☐ *PADRÕES*

☐ *Segue PowerShell Style Guide?*

☐ *PSScriptAnalyzer passa sem avisos?*

☐ *Código está formatado consistentemente?*

☐ *Verbos aprovados são usados?*

☐ *COMPATIBILIDADE*

☐ *Versão mínima do PowerShell está especificada?*

☐ *Módulos requeridos estão listados?*

☐ *Funciona em diferentes SO (se aplicável)?*

#>

 # Executar PSScriptAnalyzer

Invoke-ScriptAnalyzer -Path ".\MeuScript.ps1" -Severity Warning, Error

 # Executar testes Pester

Invoke-Pester -Path ".\Tests" -Output Detailed

 # Verificar formatação

 # Use extensão PowerShell do VS Code com formatação automática

6.2.6 README Template

Nome do Projeto

Breve descrição do projeto em uma ou duas frases.

Badges

Índice

Sobre

Descrição detalhada do projeto:

- O que ele faz
- Por que foi criado
- Principais funcionalidades

Requisitos

- PowerShell 5.1 ou superior
- Windows 10/Windows Server 2016 ou superior
- Módulos necessários:
    - ActiveDirectory
    - SqlServer

Instalação

Instalação Manual

 # Clone o repositório

git clone https://github.com/usuario/projeto.git

 # Navegue até o diretório

cd projeto

 # Importe o módulo

Import-Module .\ModuleName.psd1

Instalação via PowerShell Gallery

Install-Module -Name ModuleName

Uso

Básico

 # Exemplo de uso básico

Get-Something -Name "Valor"

Avançado

 # Exemplo de uso avançado

Get-Something -Name "Valor" -Filter { $_.Property -eq "Filtro" } |

  Set-Something -NewValue "NovoValor"

Exemplos

Exemplo 1: Cenário Comum

 # Descrição do que este exemplo faz

Get-Something -Parameter1 "Valor1" -Parameter2 "Valor2"

 # Saída esperada:

 # Nome Valor Status

 # ---- ----- ------

 # Item1 100 Ativo

Exemplo 2: Processamento em Lote

 # Processar múltiplos itens

Get-Content .\lista.txt | ForEach-Object {

```
    Get-Something -Name $_
}
```

Documentação

Documentação completa disponível em:

- Wiki do Projeto
- Documentação de API

Funções Principais

- Get-Something - Obtém recursos
- Set-Something - Configura recursos
- New-Something - Cria novos recursos
- Remove-Something - Remove recursos

Contribuindo

Contribuições são bem-vindas! Por favor:

1. Fork o projeto
2. Crie uma branch para sua feature (git checkout -b feature/NovaFuncionalidade)
3. Commit suas mudanças (git commit -m 'Adiciona nova funcionalidade')
4. Push para a branch (git push origin feature/NovaFuncionalidade)
5. Abra um Pull Request

Guidelines

- Siga o PowerShell Style Guide
- Adicione testes para novas funcionalidades
- Atualize a documentação
- Execute PSScriptAnalyzer antes de submeter

Licença

Este projeto está licenciado sob a Licença MIT - veja o arquivo LICENSE para detalhes.

Autores

- **João Silva** - *Trabalho Inicial* - GitHub

Agradecimentos

- Agradecimento especial aos contribuidores
- Inspiração: Projeto X

Changelog

Veja CHANGELOG.md para histórico de versões.

Suporte

Para problemas, dúvidas ou sugestões:

- Abra uma Issue
- Email: suporte@empresa.com
- Slack: #powershell-help

Conclusão da Seção 6

Nesta seção, exploramos as melhores práticas e aspectos de segurança essenciais para desenvolvimento profissional em PowerShell:

1. **Gestão de Credenciais**:
    - Métodos seguros de armazenamento
    - Export-Clixml, SecureString, Credential Manager
    - Azure Key Vault para ambientes enterprise
2. **Segurança em Scripts**:
    - Validação e sanitização de entrada
    - Execution Policy e assinatura digital
    - Logging e auditoria
    - PowerShell Remoting seguro
    - Just Enough Administration (JEA)
3. **Padronização**:
    - Convenções de nomenclatura
    - Estrutura de scripts e módulos
    - Versionamento semântico
4. **Documentação**:

- Comment-based help completo
- Comentários inline efetivos
- README profissional
- Code review checklist

Com essas práticas, você pode criar scripts PowerShell seguros, profissionais, manuteníveis e bem documentados que atendam aos mais altos padrões da indústria.