

Guia prático de PowerShell

Com exemplos e scripts comentados



ANDRÉ FORLIN DOSCIATI

Sumário

Fundamentos do PowerShell.....	2
1.2. Instalação e Configuração	2
Ubuntu:	3
2 CentOS.....	3
1.3. Interface do PowerShell	3
3.0 Comandos e Scripts	10
4. Administração de Sistemas com PowerShell.....	17
5. Gerenciamento de processos.....	20
Cenários de Uso	22
6. Scripts e Automação com PowerShell	23
6.5 Backup de Projetos de Desenvolvimento	26
Automatizar a Limpeza de Arquivos Temporários.....	27
Sincronização de Diretórios com Robocopy	29
Criação de Ponto de Restauração do Sistema	29
Verificação e Correção de Erros no Disco.....	30
Exportação de Políticas de Segurança para Backup	31
Backup do registro do Windows	31
Restauração do Backup do registro do Windows.....	31
Recursos Adicionais	32

Fundamentos do PowerShell

1.1. O que é PowerShell?

Histórico e Evolução

PowerShell é uma plataforma de automação de tarefas e gerenciamento de configuração da Microsoft, que consiste em um shell de linha de comando e uma linguagem de script. Foi inicialmente lançado em 2006 como Windows PowerShell, sendo projetado para automação de administração de sistemas e tarefas de gerenciamento. A evolução do PowerShell ao longo dos anos inclui a transição de um produto exclusivo para Windows para uma ferramenta multiplataforma com a introdução do PowerShell Core em 2016, que é baseado no .NET Core, permitindo sua execução em Linux e macOS, além do Windows. Em 2021, a Microsoft consolidou as versões em uma única marca, simplesmente chamada PowerShell, abandonando o "Core" do nome.

Comparação com Outras Ferramentas de Linha de Comando

Comparado com outras ferramentas de linha de comando como o CMD (Prompt de Comando) do Windows e o Bash no Linux, o PowerShell se destaca por várias razões:

- **CMD:** Enquanto o CMD é uma ferramenta simples de linha de comando com funcionalidades limitadas e uma sintaxe básica, o PowerShell oferece uma linguagem de script poderosa, suporte a objetos complexos e integração profunda com o sistema operacional.
- **Bash:** Embora o Bash seja extremamente popular no ambiente Linux e possua uma rica coleção de scripts e ferramentas, o PowerShell traz vantagens na manipulação de objetos e interoperabilidade com tecnologias Microsoft, além de ser agora multiplataforma.

Versões do PowerShell

- **Windows PowerShell:** As primeiras versões foram desenvolvidas exclusivamente para o Windows, com versões principais incluindo 1.0, 2.0, 3.0, 4.0 e 5.0, cada uma adicionando novas funcionalidades e melhorando a linguagem.
- **PowerShell Core:** A partir da versão 6.0, o PowerShell se tornou multiplataforma, baseado no .NET Core, e foi renomeado para PowerShell Core.
- **PowerShell 7:** Lançado como a versão mais recente e atualmente chamada apenas PowerShell, esta versão combina as capacidades do Windows PowerShell com as do PowerShell Core, sendo a versão recomendada para novos projetos e scripts.

1.2. Instalação e Configuração

Como Instalar em Diferentes Sistemas Operacionais

Windows

1. **Windows PowerShell:** Pré-instalado nas versões modernas do Windows. Para versões antigas, pode ser atualizado via Windows Management Framework.
2. **PowerShell 7:**
 - Baixe o instalador do [GitHub](#).
 - Execute o instalador e siga as instruções.

macOS

1. Abra o Terminal.

2. Instale o Homebrew se ainda não tiver: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
3. Instale o PowerShell com o comando: `brew install --cask powershell`
4. Execute o PowerShell com o comando: `pwsh`

Linux

Ubuntu:

- Abra o Terminal.
- Execute os comandos:

```
sudo apt-get update
sudo apt-get install -y wget apt-transport-https software-properties-common
wget -q https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
sudo apt-get install -y powershell
```

- Execute o PowerShell com o comando: `pwsh`

2 CentOS:

- Abra o Terminal.
- Execute os comandos:

```
sudo yum install -y wget
wget -q https://packages.microsoft.com/config/rhel/7/packages-microsoft-prod.rpm
sudo rpm -Uvh packages-microsoft-prod.rpm
sudo yum install -y powershell
```

- Execute o PowerShell com o comando: `pwsh`

Configurações Básicas e Perfil de Usuário

- **Perfis do PowerShell:** Arquivos de script que são executados automaticamente quando o PowerShell é iniciado. Localizações típicas:
- Windows:
`C:\Users\<username>\Documents\PowerShell\Microsoft.PowerShell_profile.ps1`
- macOS/Linux: `~/.config/powershell/Microsoft.PowerShell_profile.ps1`

Para criar ou editar um perfil:

1. Abra o PowerShell.
2. Verifique se o perfil existe com: `Test-Path $PROFILE`
3. Se não existir, crie o perfil com: `New-Item -Path $PROFILE -Type File -Force`
4. Edite o perfil com: `notepad $PROFILE` (Windows) ou `nano $PROFILE` (macOS/Linux).
5. Adicione comandos, alias ou funções que deseja executar ao iniciar o PowerShell.

1.3. Interface do PowerShell

PowerShell Console vs. PowerShell ISE vs. Visual Studio Code

PowerShell Console

- Interface padrão de linha de comando.
- Utilizada principalmente para execução rápida de comandos e scripts.

PowerShell ISE (Integrated Scripting Environment)

- Ambiente de desenvolvimento integrado específico para PowerShell.
- Oferece recursos como destaque de sintaxe, depuração e execução de scripts.
- Ideal para escrever e testar scripts PowerShell.

Visual Studio Code

- Editor de código leve e extensível, suportando múltiplas linguagens.
- Possui extensão PowerShell para suporte completo à linguagem.
- Recomendado para desenvolvimento avançado de scripts e módulos PowerShell devido a suas capacidades de integração e personalização.

Navegação Básica na Interface

PowerShell Console e ISE

- **Executar Comandos:** Digite um comando e pressione Enter.
- **Histórico de Comandos:** Use as setas para cima e para baixo para navegar pelos comandos anteriores.
- **Autocompletar:** Use a tecla Tab para autocompletar comandos e parâmetros.

Visual Studio Code

- **Abrir Terminal:** Ctrl + ' para abrir o terminal integrado.
- **Executar Scripts:** Abra o arquivo .ps1 e pressione F5 para executar.
- **Extensão PowerShell:** Instale a extensão PowerShell para suporte avançado, como autocompletar, depuração e formatação de código.

2.1. Cmdlets e Sintaxe Básica

Estrutura dos Cmdlets (Verbo-Substantivo)

Os cmdlets (pronunciado "command-lets") são os comandos utilizados no PowerShell. Eles seguem uma convenção de nomenclatura consistente que facilita a leitura e a compreensão dos scripts. Cada cmdlet é estruturado no formato "Verbo-Substantivo".

Verbo-Substantivo

Verbo: Representa a ação a ser realizada. Alguns verbos comuns incluem → [Get](#), [Set](#), [New](#), [Remove](#), [Start](#), [Stop](#).

Substantivo: Indica o objeto no qual a ação será realizada. Exemplo: → [Process](#), [Service](#), [Item](#), [content](#).

Exemplos de Cmdlets:

- [Get-Process](#): Obtém informações sobre processos em execução.
- [Set-Item](#): Define o valor de um item.
- [New-Item](#): Cria um novo item, como um arquivo ou pasta.
- [Remove-Service](#): Remove um serviço.

Execução de Cmdlets Básicos

Get-Help O cmdlet Get-Help é utilizado para obter informações detalhadas sobre outros cmdlets e suas sintaxes. É uma ferramenta essencial para aprender a usar cmdlets e entender suas opções.

Exemplo:

[Get-Help Get-Process](#)

Parâmetros Úteis:

- -Detailed: Fornece uma descrição detalhada do cmdlet.
- -Examples: Exibe exemplos de como usar o cmdlet.
- -Full: Exibe toda a documentação disponível para o cmdlet.

[Get-Help Get-Process -Examples](#)

Get-Command O cmdlet Get-Command lista todos os cmdlets, funções, aliases e scripts disponíveis na sessão atual do PowerShell. É útil para descobrir comandos que podem ser usados.

- **Exemplo:**

[Get-Command](#)

- **Filtrar por Verbo:**

[Get-Command -Verb Get](#)

- **Filtrar por Substantivo:**

[Get-Command -Noun Process](#)

Get-Member O cmdlet Get-Member exibe as propriedades e métodos dos objetos. É usado para explorar e entender os tipos de objetos que os cmdlets retornam.

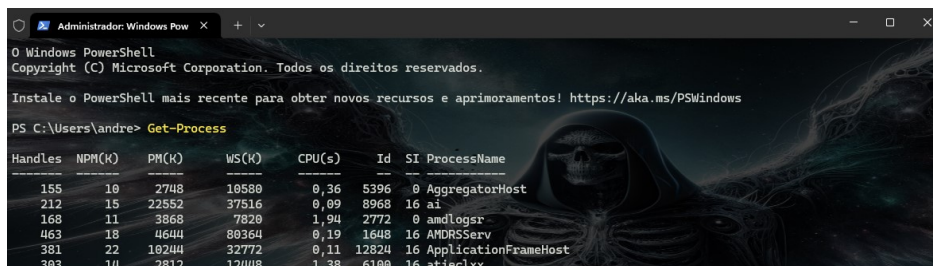
- **Exemplo:**

[Get-Process](#) | [Get-Member](#)

- **Descrição:**
 - [Get-Process](#) obtém uma lista de processos em execução.
 - | (pipe) passa a saída de [Get-Process](#) como entrada para [Get-Member](#).
 - [Get-Member](#) exibe as propriedades (como **Id**, **Name**, **CPU**) e métodos (como **Kill**, **Start**) dos objetos de processo.

Exemplos Práticos

Exemplo 1: Listar todos os processos em execução



Handles	NPM(K)	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
155	10	2748	10580	0,36	5396	0	AggregatordHost
212	15	22552	37516	0,09	8968	16	ai
168	11	3868	7820	1,94	2772	0	amdlogsr
463	18	4644	80364	0,19	1648	16	AMDRSServ
381	22	10244	32772	0,11	12824	16	ApplicationFrameHost
393	14	2812	12448	1,38	6100	16	atieclxx

Exemplo 2: Obter ajuda sobre o cmdlet Get-Process

```
Administrador: Windows Pow x + v
PS C:\Users\andre> Get-Help Get-Process -Detailed

NOME
    Get-Process

SINTAXE
    Get-Process [[-Name] <string[]>] [<CommonParameters>]

    Get-Process [[-Name] <string[]>] [<CommonParameters>]
```

Exemplo 3: Listar todos os cmdlets que começam com "Get"

```
Administrador: Windows Pow x + v
PS C:\Users\andre> Get-Command -Verb Get

CommandType Name Version Source
-----
Alias Get-AppPackage 2.0.1.0 Appx
Alias Get-AppPackageAutoUpdateSettings 2.0.1.0 Appx
Alias Get-AppPackageDefaultVolume 2.0.1.0 Appx
Alias Get-AppPackageLastError 2.0.1.0 Appx
Alias Get-AppPackageLog 2.0.1.0 Appx
Alias Get-AppPackageManifest 2.0.1.0 Appx
Alias Get-AppPackageVolume 2.0.1.0 Appx
```

Exemplo 4: Exibir as propriedades e métodos dos objetos retornados pelo cmdlet Get-Service

```
Administrador: Windows Pow x + v
PS C:\Users\andre> Get-Service | Get-Member

TypeName: System.ServiceProcess.ServiceController

Name MemberType Definition
-----
Name AliasProperty Name = ServiceName
RequiredServices AliasProperty RequiredServices = ServicesDependedOn
Disposed Event System.EventHandler Disposed(System.Object, System.EventArgs)
Close Method void Close()
Continue Method void Continue()
CreateObjRef Method System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
```

2.2. Objetos e Pipeline

Modelo Baseado em Objetos

Diferentemente de outras ferramentas de linha de comando que retornam texto simples, o PowerShell utiliza um modelo baseado em objetos. Isso significa que os cmdlets do PowerShell retornam objetos .NET, proporcionando uma maneira rica e estruturada de manipular dados.

Objetos .NET

- **Propriedades:** Atributos que descrevem as características de um objeto. Por exemplo, um objeto Process pode ter propriedades como Id, Name e CPU.
- **Métodos:** Ações que podem ser realizadas em um objeto. Por exemplo, um objeto Process pode ter métodos como Kill() e Start().

Exemplo de Retorno de Objeto

O cmdlet `Get-Process` retorna objetos `Process` que têm várias propriedades e métodos associados.

```
$process = Get-Process -Name "notepad"
```

Para acessar as propriedades do objeto retornado:

```
$process.Id  
$process.Name  
$process.CPU
```

Pipeline

O pipeline é um dos conceitos mais poderosos do PowerShell. Ele permite encadear cmdlets de maneira que a saída de um cmdlet seja passada como entrada para o próximo cmdlet. Isso facilita o processamento e manipulação de dados de maneira sequencial.

Operador de Pipeline (|)

- O operador de pipeline (|) é usado para conectar cmdlets.
- A saída de um cmdlet é passada como entrada para o cmdlet seguinte no pipeline.

Exemplo Simples de Pipeline

- Listar os processos em execução e selecionar apenas os processos que estão consumindo mais de 100MB de memória.

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

- **Get-Process**: Obtém a lista de processos em execução.
- **|**: Passa a saída do **Get-Process** para o próximo cmdlet.
- **Where-Object { \$_.WorkingSet -gt 100MB }**: Filtra os processos com uso de memória superior a 100MB. O **\$_** representa o objeto atual no pipeline.

Exemplo de Pipeline com Vários Cmdlets

- Obter os nomes de todos os serviços em execução e ordená-los alfabeticamente.

```
Get-Service | Where-Object { $_.Status -eq 'Running' } | Sort-Object -Property Name | Select-Object -Property Name
```

- **Get-Service**: Obtém a lista de serviços.
- **Where-Object { \$_.Status -eq 'Running' }**: Filtra os serviços que estão em execução.
- **Sort-Object -Property Name**: Ordena os serviços pelo nome.
- **Select-Object -Property Name**: Seleciona apenas a propriedade Name dos serviços.

Trabalhando com Propriedades e Métodos

- Para exibir as propriedades de um objeto no pipeline, use o cmdlet **Select-Object**.

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

Para invocar métodos em objetos no pipeline, use o cmdlet **ForEach-Object**.

```
Get-Process | ForEach-Object {  
    [PSCustomObject]@{  
        ProcessName = $_.Name  
        PID = $_.Id  
        MemoryUsageMB = [math]::Round($_.WorkingSet64 / 1MB, 2)  
    }  
} | Format-Table -AutoSize
```

Exemplos Práticos

Exemplo 1: Obter detalhes de processos específicos

```
Get-Process -Name "explorer" | Select-Object -Property Name, Id, CPU, StartTime
```

Exemplo 2: Listar serviços e exibir apenas o nome e status

```
Get-Service | Select-Object -Property Name, Status
```

Exemplo 3: Filtrar arquivos em um diretório por tamanho e ordenar por nome

```
Get-ChildItem -Path "C:\Logs" | Where-Object { $_.Length -gt 1MB } | Sort-Object -Property Name
```

Exemplo 4: Reiniciar serviços que estão parados

```
Get-Service | Where-Object { $_.Status -eq 'Stopped' } | ForEach-Object { $_.Start() }
```

2.3. Variáveis e Tipos de Dados

Declaração e Uso de Variáveis

No PowerShell, variáveis são usadas para armazenar dados temporariamente durante a execução de scripts e comandos. As variáveis são prefixadas com o símbolo \$.

Declaração de Variáveis

Para declarar uma variável, basta prefixar o nome da variável com \$ e atribuir um valor a ela usando o operador =.

Exemplos:

```
$nome = "João"
```

```
$idade = 30
```

Uso de Variáveis

- Depois de declarada, uma variável pode ser utilizada em comandos e scripts.
- As variáveis podem armazenar diferentes tipos de dados, como strings, números, arrays e hashtables.

Exemplo de Uso:

```
$nome = "João"
```

```
$idade = 30
```

```
$saudacao = "Olá, meu nome é $nome e eu tenho $idade anos."
```

```
Write-Output $saudacao
```

Tipos de Dados Comuns

Strings

- Strings são usadas para armazenar texto.
- Podem ser delimitadas por aspas simples (') ou aspas duplas (").
- Aspas duplas permitem a interpolação de variáveis.

Exemplos:

```
$string1 = "Texto com variável: $nome"
```

```
$string2 = 'Texto sem interpolação de variável: $nome'
```

Inteiros

- Inteiros são números sem casas decimais.
- Usados para armazenar valores numéricos.

Exemplo:

```
$numero = 42
```

Arrays

- Arrays são coleções de elementos que podem ser de diferentes tipos.
- Podem ser criados usando o operador @.

Exemplo:

```
$array = @("um", 2, 3.0, $nome)
```

```
$array[0] # Acessa o primeiro elemento
```

Hashtables

- Hashtables são coleções de pares chave-valor.
- Usadas para armazenar dados de forma associativa.

Exemplo:

```
$hashtable = @{
```

```
  "Nome" = "João"
```

```
  "Idade" = 30
```

```
  "Cidade" = "São Paulo"
```

```
}
```

```
$hashtable["Nome"] # Acessa o valor associado à chave "Nome"
```

Exemplos Práticos

Exemplo 1: Declaração e uso de variáveis de diferentes tipos

```
# String
```

```
$nome = "Ana"
```

```
# Inteiro
```

```
$idade = 25
```

```
# Array
```

```
$numeros = @(1, 2, 3, 4, 5)
```

```
# Hashtable
```

```
$info = @{
```

```
  "Nome" = "Ana"
```

```
  "Idade" = 25
```

```
  "Cidade" = "Rio de Janeiro"
```

```
}
```

```
# Uso das variáveis
```

```
Write-Output "Nome: $nome, Idade: $idade"
```

```
Write-Output "Primeiro número: $($numeros[0])"
```

```
Write-Output "Cidade: $($info["Cidade"])"
```

Exemplo 2: Manipulação de Arrays e Hashtables

```
# Array de strings
```

```
$frutas = @("Maçã", "Banana", "Laranja")
```

```
# Adicionar um item ao array
```

```
$frutas += "Uva"
```

```
# Exibir o array
```

```
$frutas
```

```
# Hashtable de informações de um livro
```

```
$livro = @{
```

```

"Título" = "1984"
"Autor" = "George Orwell"
"Ano" = 1949
}
# Adicionar uma nova chave-valor
$livro["Editora"] = "Secker & Warburg"
# Exibir a hashtable
$livro

```

3.0 Comandos e Scripts

Introdução aos Comandos e Scripts no PowerShell

No PowerShell, os comandos e scripts são os blocos de construção fundamentais para automatizar tarefas, gerenciar sistemas e realizar operações complexas de forma eficiente. Comandos são instruções individuais que você executa no shell, enquanto scripts são coleções de comandos agrupados em um arquivo para serem executados como uma unidade.

Comandos

Cmdlets

Cmdlets são comandos embutidos no PowerShell, projetados para executar uma tarefa específica. Como discutido anteriormente, os cmdlets seguem a convenção de nomenclatura Verbo-Substantivo e são usados para uma ampla gama de tarefas, como gerenciamento de arquivos, processos, serviços, registros e mais.

Exemplos de Cmdlets Comuns:

- **Get-Process** : Obtém informações sobre processos em execução.
- **Stop-Service**: Interrompe um serviço.
- **Set-Item**: Define o valor de um item.
- **Get-EventLog**: Obtém entradas de um log de eventos.

Alias

Alias são nomes alternativos para cmdlets ou comandos. Eles permitem que você use versões abreviadas ou mais familiares de comandos, o que pode ser útil para quem vem de outros ambientes de linha de comando.

Exemplos de Alias:

- **ls**: Alias para **Get-ChildItem**.
- **cd**: Alias para **Set-Location**.
- **cp**: Alias para **Copy-Item**.
- **rm**: Alias para **Remove-Item**.

Você pode ver todos os alias disponíveis usando o comando:

```
get-alias
```

Funções

Funções são blocos de código reutilizáveis que você pode definir dentro do PowerShell para executar tarefas específicas. Elas permitem encapsular lógica complexa em um comando simples que pode ser reutilizado.

Exemplo de Definição de Função:

```

function Get-Greeting {
    param (

```

```

    [string]$Name
)
"Hello, $Name! Welcome to PowerShell."
}

```

```

# Usar a função
Get-Greeting -Name "Maria"

```

Scripts

Scripts no PowerShell são arquivos de texto que contêm uma sequência de comandos e são salvos com a extensão .ps1. Eles permitem a automação de tarefas complexas e repetitivas, proporcionando um meio de executar vários comandos de uma vez.

Criando e Executando Scripts

Passos para Criar um Script:

1. Abra um editor de texto (como Notepad ou Visual Studio Code).
2. Escreva os comandos que deseja executar.
3. Salve o arquivo com a extensão .ps1 (por exemplo, meuScript.ps1).

```

# Script: greetings.ps1
$nome = "Carlos"
$saudacao = "Olá, $nome! Bem-vindo ao PowerShell."
Write-Output $saudacao

```

```

# Listar processos em execução
Get-Process

```

Executando o Script: Para executar um script, abra o PowerShell e navegue até o diretório onde o script está salvo, então execute o script usando: **.\greetings.ps1**

Configurações de Execução

O PowerShell possui uma política de execução de scripts que determina quais scripts podem ser executados em um sistema. As políticas de execução mais comuns são:

- **Restricted:** Nenhum script pode ser executado. Esta é a configuração padrão.
- **AllSigned:** Somente scripts assinados digitalmente podem ser executados.
- **RemoteSigned:** Scripts baixados da internet precisam ser assinados digitalmente, mas scripts locais podem ser executados sem assinatura.
- **Unrestricted:** Todos os scripts podem ser executados, mas scripts baixados da internet exigem confirmação.

Você pode verificar e alterar a política de execução usando os seguintes comandos:

Verificar a Política de Execução:

```
Get-ExecutionPolicy
```

Alterar a Política de Execução (por exemplo, para RemoteSigned):

```
Set-ExecutionPolicy RemoteSigned
```

Parâmetros e Argumentos

Scripts e funções no PowerShell podem aceitar parâmetros e argumentos, permitindo que você passe valores dinâmicos e personalize a execução.

Exemplo de Script com Parâmetros:

```
# Script: backup.ps1
param (
    [string]$SourcePath,
    [string]$DestinationPath
)
```

```
Copy-Item -Path $SourcePath -Destination $DestinationPath -Recurse
Write-Output "Backup completed from $SourcePath to $DestinationPath."
```

Para executar o script com parâmetros:

```
.\backup.ps1 -SourcePath "C:\Dados" -DestinationPath "D:\Backup"
```

O que este script faz:

- **param:** Define os parâmetros que o script pode receber quando for executado.
- **[string]\$SourcePath:** Este parâmetro representa o caminho de origem (source path), ou seja, o diretório ou arquivo que será copiado. O tipo de dado é string, indicando que espera-se um caminho de arquivo ou pasta como uma cadeia de caracteres.
- **[string]\$DestinationPath:** Este parâmetro representa o caminho de destino (destination path), para onde os arquivos ou diretórios serão copiados. Também é uma string.
- **Copy-Item:** Cmdlet usado para copiar arquivos e diretórios.
- **-Path \$SourcePath:** Especifica o caminho de origem dos arquivos ou diretórios que serão copiados.
- **-Destination \$DestinationPath:** Especifica o caminho de destino para onde os arquivos ou diretórios serão copiados.
- **-Recurse:** Este parâmetro é utilizado para copiar o conteúdo de forma recursiva, ou seja, ele copia todos os subdiretórios e seus conteúdos dentro do diretório de origem.
- **Write-Output:** Cmdlet que envia uma mensagem para a saída padrão, geralmente exibida no console.
- **Mensagem:** A mensagem "Backup completed from \$SourcePath to \$DestinationPath." é exibida para informar ao usuário que a cópia de segurança foi concluída com sucesso. Os valores de \$SourcePath e \$DestinationPath são interpolados na string para mostrar os caminhos específicos usados na operação.

Estruturas de Controle em Scripts

Scripts no PowerShell podem incluir estruturas de controle como loops (for, foreach, while) e condicionais (if, else, switch) para executar operações repetitivas e tomar decisões.

Exemplo de Estrutura Condicional:

```
$valor = 10

if ($valor -gt 5) {
    Write-Output "O valor é maior que 5."
} else {
    Write-Output "O valor é 5 ou menor."
}
```

Exemplo de Loop:

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

Exemplos Práticos de Scripts

Exemplo 1: Script de Backup Automático

```
param (
```

```
[string]$SourceDir,
[string]$BackupDir
)

if (-not (Test-Path $BackupDir)) {
    New-Item -ItemType Directory -Path $BackupDir
}

Copy-Item -Path $SourceDir -Destination $BackupDir -Recurse
Write-Output "Backup from $SourceDir to $BackupDir completed."
```

Exemplo 2: Script para Verificar o Uso de CPU de Processos

```
$processos = Get-Process | Where-Object { $_.CPU -gt 100 }

foreach ($processo in $processos) {
    Write-Output "Processo: $($processo.Name) - CPU: $($processo.CPU)"
}
```

3.1. Trabalhando com Arquivos e Diretórios

O PowerShell oferece uma série de cmdlets e funcionalidades para manipular arquivos e diretórios. Com esses comandos, você pode criar, copiar, mover, excluir, e listar arquivos e pastas de maneira eficiente.

Navegação em Diretórios

- **Get-Location** (alias: **pwd**): Exibe o diretório atual em que você está navegando.

Get-Location

- **Set-Location** (alias: **cd**): Altera o diretório atual.

Set-Location -Path "C:\Users\SeuUsuario\Documentos"

- **Push-Location** e **Pop-Location**: Permitem alternar temporariamente para um diretório diferente e retornar ao original.

Push-Location "C:\Windows"

Fazer algo no diretório C:\Windows

Pop-Location

```
PS C:\Windows\System32> Push-Location "C:\Windows"
PS C:\Windows> Pop-Location
PS C:\Windows\System32>
```

Manipulação de Arquivos e Diretórios

- **Get-ChildItem** (alias: **ls**, **dir**): Lista arquivos e diretórios em um local específico.

Get-ChildItem -Path "C:\Users\andre\Documentos"

- **New-Item**: Cria novos arquivos ou diretórios.

Criar um diretório

```
New-Item -Path "C:\Users\SeuUsuario\Documentos\NovaPasta" -ItemType Directory
```

Criar um arquivo de texto

```
New-Item -Path "C:\Users\SeuUsuario\Documentos\NovoArquivo.txt" -ItemType File
```

- **Copy-Item (alias: cp):** Copia arquivos ou diretórios.

```
Copy-Item -Path "C:\Fonte\Arquivo.txt" -Destination "C:\Destino\Arquivo.txt"
```

- **Move-Item (alias: mv):** Move ou renomeia arquivos e diretórios.

```
Move-Item -Path "C:\Fonte\Arquivo.txt" -Destination "C:\Destino\Arquivo.txt"
```

- **Remove-Item (alias: rm, del):** Remove (exclui) arquivos ou diretórios.

```
Remove-Item -Path "C:\Users\SeuUsuario\Documentos\ArquivoParaExcluir.txt"
```

Leitura e Escrita de Arquivos

- **Get-Content (alias: gc):** Lê o conteúdo de um arquivo.

```
Get-Content -Path "C:\Users\SeuUsuario\Documentos\Arquivo.txt"
```

- **Set-Content:** Escreve ou substitui o conteúdo de um arquivo.

```
Set-Content -Path "C:\Users\SeuUsuario\Documentos\Arquivo.txt" -Value "Novo conteúdo"
```

- **Add-Content:** Adiciona texto ao final de um arquivo existente.

```
Add-Content -Path "C:\Users\SeuUsuario\Documentos\Arquivo.txt" -Value "Texto adicional"
```

- **Out-File:** Redireciona a saída de um comando para um arquivo.

```
Get-Process | Out-File -FilePath C:\Users\SeuUsuario\Documentos\Processos.txt"
```

3.2. Controle de Fluxo

O controle de fluxo no PowerShell permite que você controle a lógica dos scripts e comandos, definindo como e quando determinadas seções de código serão executadas.

Estruturas Condicionais

- **if, elseif, else:** Executa comandos com base em condições.

```
$idade = 20
```

```
if ($idade -ge 18) {
    Write-Output "Você é maior de idade."
} elseif ($idade -lt 18 -and $idade -ge 12) {
    Write-Output "Você é adolescente."
} else {
    Write-Output "Você é uma criança."
}
```

- **switch:** Avalia uma expressão uma vez e executa o bloco de código correspondente.

```
$diaDaSemana = "Segunda-feira"
```

```
switch ($diaDaSemana) {
    "Segunda-feira" { Write-Output "Início da semana." }
    "Sexta-feira" { Write-Output "Quase fim de semana." }
    "Sábado" { Write-Output "Fim de semana!" }
    Default { Write-Output "Dia normal." }
}
```

Loops

- **for:** Executa um bloco de código repetidamente com base em uma condição.

```
for ($i = 0; $i -lt 5; $i++) {
    Write-Output "Contagem: $i"
}
```

- **foreach**: Itera sobre cada item em uma coleção.

```
$nomes = @("Ana", "Bruno", "Carlos")

foreach ($nome in $nomes) {
    Write-Output "Olá, $nome!"
}
```

- **while**: Executa um bloco de código enquanto uma condição for verdadeira.

```
$contador = 0

while ($contador -lt 3) {
    Write-Output "Contagem: $contador"
    $contador++
}
```

- **do...while**: Similar ao while, mas a condição é verificada após a execução do bloco.

```
$contador = 0

do {
    Write-Output "Contagem: $contador"
    $contador++
} while ($contador -lt 3)
```

Manipulação de Exceções

- **try, catch, finally**: Usados para capturar e tratar erros durante a execução de comandos.

```
try {
    $resultado = 1 / 0 # Gera um erro de divisão por zero
} catch {
    Write-Output "Erro capturado: $_"
} finally {
    Write-Output "Execução concluída."
}
```

3.3. Funções e Scripts

Funções e scripts são fundamentais para reutilizar código e automatizar tarefas repetitivas no PowerShell.

Funções

As funções permitem encapsular blocos de código que podem ser chamados repetidamente em um script ou sessão. Elas podem receber parâmetros e retornar valores.

Definindo Funções:

```
function Saudacao {
    param (
        [string]$Nome,
        [int]$Idade
    )

    return "Olá, $Nome! Você tem $Idade anos."
}
```



```
# Chamando a função
$mensagem = Saudacao -Nome "Lucas" -Idade 30
Write-Output $mensagem
```

Funções Avançadas: Funções avançadas no PowerShell são semelhantes a cmdlets e podem utilizar recursos como parâmetros obrigatórios, validação de entrada e suporte a pipelines.

```
function Get-MaiorNumero {
    [CmdletBinding()]
    param (
        [Parameter(Mandatory=$true)]
        [int[]]$Numeros
    )

    return ($Numeros | Measure-Object -Maximum).Maximum
}
```

```
# Usando a função avançada
$maiorNumero = Get-MaiorNumero -Numeros @(10, 20, 30)
Write-Output "O maior número é: $maiorNumero"
```

Scripts

Scripts no PowerShell são arquivos que contêm uma sequência de comandos e funções, permitindo automatizar tarefas complexas.

Criando e Executando Scripts:

- Um script é simplesmente um arquivo .ps1 que contém comandos do PowerShell.
- Para criar um script, abra um editor de texto, escreva os comandos e salve o arquivo com a extensão .ps1.

Exemplo de Script:

```
# Script: backup.ps1
param (
    [string]$SourceDir,
    [string]$BackupDir
)

if (-not (Test-Path $BackupDir)) {
    New-Item -ItemType Directory -Path $BackupDir
}

Copy-Item -Path $SourceDir -Destination $BackupDir -Recurse
Write-Output "Backup realizado de $SourceDir para $BackupDir"
```

Executando o Script:

```
.\backup.ps1 -SourceDir "C:\Dados" -BackupDir "D:\Backup"
```

Passando Parâmetros para Scripts: Scripts podem aceitar parâmetros para permitir que sejam mais flexíveis e reutilizáveis.

```
# Script: saudacao.ps1
param (
    [string]$Nome = "Visitante",
    [int]$Idade = 25
)

Write-Output "Olá, $Nome! Você tem $Idade anos."
```

Executando com Parâmetros:

```
.\saudacao.ps1 -Nome "João" -Idade 40
```

Modularizando Scripts

À medida que seus scripts crescem em tamanho e complexidade, é uma boa prática modularizar o código, dividindo-o em funções e scripts menores e reutilizáveis.

Importando Funções de Outros Scripts:

```
# Importa as funções de um script externo  
. \funcoes.ps1
```

```
# Agora você pode usar as funções importadas  
MinhaFuncao -Parametro1 "Valor"
```

4. Administração de Sistemas com PowerShell

O PowerShell é uma ferramenta poderosa para a administração de sistemas, proporcionando aos administradores uma maneira eficiente e flexível de gerenciar e automatizar uma vasta gama de tarefas de administração de sistemas. Abaixo, exploramos os principais tópicos e comandos que capacitam os administradores a gerenciar sistemas Windows e ambientes multiplataforma com PowerShell.

4.1. Gerenciamento de Usuários e Grupos

Criação, Modificação e Exclusão de Usuários

- **Criar Novo Usuário:** O cmdlet `New-LocalUser` é usado para criar novos usuários locais em uma máquina Windows.

```
New-LocalUser -Name "NovoUsuario" -Password (ConvertTo-SecureString "SenhaSegura123"  
-AsPlainText -Force) -FullName "Nome Completo" -Description "Descrição do Usuário"
```

Modificar Usuário: Para modificar as propriedades de um usuário, você pode usar o cmdlet `Set-LocalUser`.

```
Set-LocalUser -Name "NovoUsuario" -FullName "Nome Completo Atualizado" -Description  
"Descrição Atualizada"
```

Excluir Usuário: Para excluir um usuário, utilize o cmdlet `Remove-LocalUser`.

```
Remove-LocalUser -Name "NovoUsuario"
```

Gerenciamento de Grupos

- **Criar Novo Grupo:** Use `New-LocalGroup` para criar grupos locais.

```
New-LocalGroup -Name "NovoGrupo" -Description "Descrição do Grupo"
```

- **Adicionar Usuário a um Grupo:** O cmdlet `Add-LocalGroupMember` adiciona um usuário a um grupo específico.

```
Add-LocalGroupMember -Group "Administradores" -Member "NovoUsuario"
```

- **Remover Usuário de um Grupo:** Para remover um usuário de um grupo, use `Remove-LocalGroupMember`.

```
Remove-LocalGroupMember -Group "Administradores" -Member "NovoUsuario"
```

- **Excluir Grupo:** Para excluir um grupo, use Remove-LocalGroup.

`Remove-LocalGroup -Name "NovoGrupo"`

4.2. Gerenciamento de Processos e Serviços

Gerenciamento de Processos

- **Listar Processos:** Use Get-Process para listar todos os processos em execução.

`Get-Process`

- **Finalizar Processos:** Para finalizar um processo específico, use Stop-Process.

`Stop-Process -Name "notepad"`

- **Iniciar Novos Processos:** Inicie um novo processo com Start-Process.

`Start-Process "notepad.exe"`

Gerenciamento de Serviços

- **Listar Serviços:** Para listar todos os serviços no sistema, use Get-Service.

`Get-Service`

Iniciar, Parar e Reiniciar Serviços:

- Iniciar um serviço:

`Start-Service -Name "NomeDoServico"`

- Parar um serviço:

`Stop-Service -Name "NomeDoServico"`

- Reiniciar um serviço:

`Restart-Service -Name "NomeDoServico"`

Alterar o Tipo de Inicialização de um Serviço: Para alterar o tipo de inicialização de um serviço (Manual, Automático, Desativado), use o cmdlet Set-Service.

`Set-Service -Name "NomeDoServico" -StartupType Automatic`

4.3. Gerenciamento de Discos e Sistema de Arquivos

Gerenciamento de Discos

- **Listar Informações sobre Discos:** Use o cmdlet Get-PhysicalDisk para obter informações sobre os discos físicos.

`Get-PhysicalDisk`

- **Gerenciamento de Volumes e Partições:** Use Get-Volume para listar todos os volumes e Get-Partition para listar as partições.

`Get-Volume`

`Get-Partition`

- **Formatar Volume:** Para formatar um volume, use Format-Volume.

`Format-Volume -DriveLetter D -FileSystem NTFS -Confirm:$false`

Gerenciamento de Sistema de Arquivos

- **Gerenciamento de Permissões (ACLs):** As listas de controle de acesso (ACLs) podem ser gerenciadas com os cmdlets Get-Acl, Set-Acl, e New-Object (para criar regras).
 - **Exibir ACLs:** (Ampliar este conteúdo com os detalhes do comando)

Get-Acl -Path "C:\windows"

- **Modificar ACLs:**

```
$acl = Get-Acl -Path "C:\PastaExemplo"  
$rule = New-Object System.Security.AccessControl.FileSystemAccessRule("Usuario",  
"FullControl", "Allow")  
$acl.SetAccessRule($rule)  
Set-Acl -Path "C:\PastaExemplo" -AclObject $acl
```

- **Compactar e Descompactar Arquivos:** Para compactar arquivos em um arquivo ZIP, use Compress-Archive, e para descompactar, use Expand-Archive.

Compress-Archive -Path "C:\Pasta*.*)" -DestinationPath "C:\Backup\Arquivo.zip"

Expand-Archive -Path "C:\Backup\Arquivo.zip" -DestinationPath C:\PastaRestaurada"

4.4. Gerenciamento de Rede

Configuração de Rede

- **Obter Informações de Rede:** Use Get-NetIPAddress para obter informações sobre os endereços IP configurados no sistema.

Get-NetIPAddress

- **Configurar Endereço IP Estático:** Use New-NetIPAddress para configurar um endereço IP estático.

New-NetIPAddress -InterfaceAlias "Ethernet" -IPAddress "192.168.1.100" -PrefixLength 24 -
DefaultGateway "192.168.1.1"

- **Configurar DNS:** Para configurar servidores DNS, use Set-DnsClientServerAddress.

Set-DnsClientServerAddress -InterfaceAlias "Ethernet" -ServerAddresses ("8.8.8.8", "8.8.4.4")

Testes de Conectividade e Diagnóstico

- **Ping e Testes de Conectividade:** Use Test-Connection para verificar a conectividade com outro dispositivo na rede.

Test-Connection -ComputerName "www.google.com" -Count 4

- **Monitoramento de Conexões de Rede:** Use Get-NetTCPConnection para monitorar as conexões TCP ativas.

Get-NetTCPConnection

4.5. Gerenciamento de Políticas de Grupo e Segurança Políticas de Grupo (GPO)

- **Obter Configurações de Políticas de Grupo:** Para listar as configurações aplicadas por GPOs, use Get-GPResultantSetOfPolicy.

Get-GPResultantSetOfPolicy -Computer "NomeDoComputador" -ReportType Html -Path
"C:\Relatorios\GPO-Relatorio.html"

Aplicar Configurações de Políticas de Grupo: Use gpupdate para forçar a aplicação das políticas de grupo.

gpupdate /force

Gerenciamento de Políticas de Segurança

- **Configuração de Políticas de Segurança:** Use Set-ExecutionPolicy para configurar a política de execução de scripts no PowerShell.

Set-ExecutionPolicy RemoteSigned

- **Monitoramento de Eventos de Segurança:** Use Get-EventLog para monitorar logs de eventos de segurança.

Get-EventLog -LogName Security -Newest 50 #Deve ser executado no terminal do PowerShell como Administrador!

4.6. Automação de Tarefas e Agendamento

Agendamento de Tarefas

- **Criar uma Tarefa Agendada:** Use New-ScheduledTask e Register-ScheduledTask para criar e registrar uma tarefa agendada.

```
$action = New-ScheduledTaskAction -Execute "PowerShell.exe" -Argument "-File  
C:\Scripts\Backup.ps1"  
$trigger = New-ScheduledTaskTrigger -Daily -At "03:00AM"  
Register-ScheduledTask -Action $action -Trigger $trigger -TaskName "BackupDiario" -  
Description "Backup diário do sistema"
```

Gerenciar Tarefas Agendadas: Use Get-ScheduledTask, Set-ScheduledTask e Unregister-ScheduledTask para gerenciar tarefas existentes.

Get-ScheduledTask -TaskName "BackupDiario"

Automação com Scripts

- **Automatizar Backups:** Crie scripts que automatizem o backup de dados importantes, executando-os regularmente através de tarefas agendadas.

```
# Script de backup  
Copy-Item -Path "C:\DadosImportantes" -Destination "D:\Backups\$(Get-Date -Format  
yyyyMMdd)" -Recurse
```

- **Monitoramento Automático:** Configure scripts que monitorem serviços críticos e enviem alertas em caso de falha.

```
# Script de monitoramento  
if (-not (Get-Service -Name "NomeDoServico" -ErrorAction SilentlyContinue).Status -eq  
'Running') {  
    Send-MailMessage -To "admin@example.com" -From "monitor@example.com" -Subject  
    "Serviço Parado" -Body "O serviço NomeDoServico parou." -SmtpServer "smtp.example.com"  
}
```

5. Gerenciamento de processos

Gerenciar processos em um sistema operacional é uma das tarefas mais comuns e essenciais para administradores de sistemas e usuários avançados. O PowerShell oferece uma série de cmdlets e funções que permitem monitorar, listar, e manipular processos de maneira eficiente.

Neste tópico, exploraremos como trabalhar com processos em execução, como utilizar o comando Tasklist, e como finalizar processos específicos.

5.1 Processos em Execução

Um processo em execução é um programa ou aplicação que está atualmente sendo executado no sistema. Cada processo é identificado por um identificador único (PID - Process Identifier) e possui várias propriedades associadas, como nome, uso de memória, tempo de CPU, entre outras.

Listar Processos em Execução

- **Get-Process:** O cmdlet Get-Process é usado para listar todos os processos que estão atualmente em execução no sistema. Ele exibe informações detalhadas sobre cada processo, incluindo o nome, PID, uso de CPU, memória, entre outros.

Exemplo:

`Get-Process`

Filtrar Processos Específicos: Você pode filtrar a lista de processos para exibir apenas aqueles que correspondem a um nome específico.

`Get-Process -Name "notepad"`

Filtrar por PID: Para obter informações sobre um processo específico usando seu PID, use o parâmetro -Id.

`Get-Process -Id 1234`

Exibir Processos em Ordem de Uso de CPU: Para listar os processos em ordem de uso de CPU, use Sort-Object.

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

Monitoramento de Processos em Tempo Real

- **Atualização Contínua de Processos:** Para monitorar os processos em tempo real, você pode utilizar um loop que atualiza constantemente a lista de processos.

```
while ($true) {  
    Get-Process  
    Start-Sleep -Seconds 5  
    Clear-Host  
}
```

5.2 Utilizando o Comando Tasklist

Embora o Get-Process seja uma ferramenta poderosa, o comando Tasklist, que é um comando embutido no Windows, oferece uma alternativa para listar processos em execução. Tasklist é comumente utilizado em scripts de linha de comando (cmd) e pode ser chamado diretamente do PowerShell.

Listar Processos com Tasklist

- **Comando Básico:** O comando básico para listar processos em execução usando Tasklist é:

`tasklist`

Filtrar Processos por Nome: Para listar processos específicos com base no nome, use o parâmetro /FI para definir um filtro.

`tasklist /FI "IMAGENAME eq chrome.exe"`

Exibir Detalhes de Memória: Você pode obter detalhes mais específicos, como o uso de memória, para processos em execução.

Processos em execução

`tasklist /FI "IMAGENAME eq chrome.exe" /FO LIST`

- **/svc:** Lista todos os serviços em cada processo.
A opção `/fi` no comando `tasklist` é usada para aplicar filtros específicos aos processos listados.
- **/fi [filtro]:** Exibe um conjunto de tarefas que correspondem ao critério especificado pelo filtro.

`tasklist /fi "cputime gt 00:10:00"`

Explicação dos Parâmetros

/fi: Especifica um filtro a ser aplicado à lista de processos.

"cputime gt 00:10:00": Define o critério do filtro:

`cputime:` Refere-se ao tempo de CPU que o processo tem usado.

`gt:` Significa "greater than" (maior que).

`00:10:00:` Especifica o valor do tempo em horas, minutos e segundos.

PID eq [número]: Filtra por ID do processo

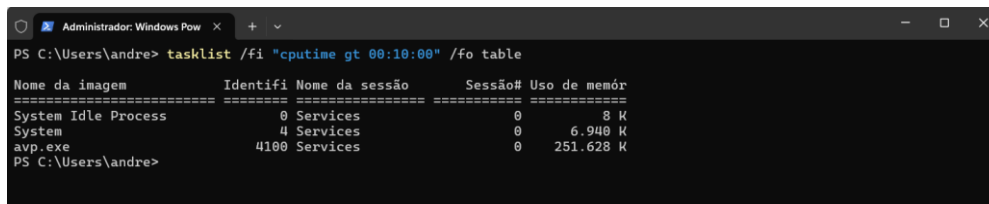
- `tasklist /fi "PID eq 1234"`

Status eq [estado]: Filtra por status (RUNNING, NOT RESPONDING).

- `tasklist /fi "status eq running"`

/fo [formato]: Especifica o formato de saída.

`tasklist /fi "cputime gt 00:10:00" /fo table`



```
PS C:\Users\andre> tasklist /fi "cputime gt 00:10:00" /fo table
Nome da imagem      Identifi Nome da sessão  Sessão#  Uso de memór
=====
System Idle Process  0 Services      0         8 K
System               4 Services      0        6.940 K
avp.exe              4100 Services    0       251.628 K
PS C:\Users\andre>
```

Cenários de Uso

Diagnóstico de Desempenho: Administradores de sistemas podem usar esse filtro para identificar processos que estão consumindo uma quantidade significativa de tempo de CPU, o que pode ajudar a identificar gargalos de desempenho ou processos que podem estar travados ou em loop.

Monitoramento de Recursos: Em servidores onde o uso de CPU é crítico, esse comando pode ajudar a monitorar e gerenciar processos que estão utilizando recursos excessivamente.

LIST: Formato de lista.

- `tasklist /fo list`
- `tasklist /fo csv`
 - `tasklist /fo csv >>task.csv` – Salva em um arquivo
 - `/v` : Exibe informações detalhadas sobre cada tarefa.
 - `tasklist /fo csv /v`

5.3 Finalizar um Processo Específico

Finalizar um processo é uma operação crítica, geralmente usada para interromper aplicativos que não estão respondendo ou para liberar recursos do sistema.

Finalizar um Processo com PowerShell

- **Stop-Process**: O cmdlet Stop-Process é usado para finalizar processos diretamente do PowerShell. Você pode especificar o processo a ser finalizado usando o nome ou o PID.
 - **Finalizar por Nome:**

Stop-Process -Name "chrome"

Finalizar por PID:

Stop-Process -Id 1234

Forçar Finalização: Para forçar a finalização de um processo que não está respondendo, use o parâmetro -Force.

Stop-Process -Name "notepad" -Force

O comando **taskkill** no Windows é usado para finalizar processos em execução. A opção **/IM** especifica o nome da imagem (processo) a ser finalizado, e a opção **/F** força a finalização do processo. Vamos explicar cada parte do comando

taskkill /IM nomeDoProcesso.exe /F.

Parâmetros

/IM [nomeDoProcesso.exe]: Especifica o nome da imagem do processo que você deseja finalizar. O nome do processo geralmente termina com .exe.

/F: Força a finalização do processo. Sem esta opção, o comando **taskkill** solicita uma finalização graciosa do processo. Com /F, ele força a finalização, independentemente de o processo estar respondendo ou não.

Cenários de Uso

Finalizar Processos Não Responsivos: Se um programa travar e não responder, você pode usar o comando **taskkill** com a opção **/F** para forçar a finalização do processo.

Gerenciamento de Processos: Administradores de sistemas podem usar este comando para gerenciar e finalizar processos em execução que estão consumindo muitos recursos ou que foram iniciados por engano.

6. Scripts e Automação com PowerShell

O PowerShell é amplamente reconhecido por suas poderosas capacidades de scripting e automação. A criação de scripts no PowerShell permite que tarefas repetitivas sejam automatizadas, resultando em maior eficiência e precisão na administração de sistemas. Nesta seção, vamos explorar os fundamentos da criação de scripts, técnicas avançadas de automação, e como aplicar práticas recomendadas para desenvolver scripts robustos e eficientes.

6.1 Script Básico para Backup Diário

Define os caminhos de origem e destino

\$source = "C:\Caminho\Para\DiretorioOrigem"

\$destination = "D:\Caminho\Para\DiretorioDestino\backup_" + (Get-Date -Format "dd-MM-yyyy")

Copia os arquivos do diretório de origem para o diretório de destino

Copy-Item -Path \$source* -Destination \$destination -Recurse -Force

Cria uma entrada no log com a data e hora atuais

\$logMessage = "Backup concluído em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss")

\$logPath = "D:\Caminho\Para\DiretorioDestino\backup_log.txt"

Add-Content -Path \$logPath -Value \$logMessage

6.1.1 Explicação Detalhada

1. Definição dos Caminhos de Origem e Destino

- **\$source:** Esta variável armazena o caminho do diretório de origem, onde os arquivos que serão copiados estão localizados.

- **\$destination**: Esta variável armazena o caminho do diretório de destino, onde os arquivos copiados serão armazenados. A variável **Get-Date -Format "dd-MM-yyyy"** é usada para adicionar a data atual ao nome do diretório de destino, criando uma pasta de backup com a data atual.

Formato de Data: A data é formatada como **"dd-MM-yyyy"**, o que resultará em algo como **"backup_22-07-2024"**, dependendo da data em que o script é executado.

2. Copiando os Arquivos do Diretório de Origem para o Diretório de Destino

- **Copy-Item**: Este é o comando PowerShell usado para copiar arquivos e diretórios.
- **-Path \$source***: Especifica o caminho de origem com um curinga (*), o que significa que todos os arquivos e subdiretórios do diretório de origem serão copiados.
- **-Destination \$destination**: Especifica o caminho de destino onde os arquivos serão copiados.
- **-Recurse**: Esta opção permite que o comando copie recursivamente todos os subdiretórios e seus conteúdos.
- **-Force**: Esta opção força a cópia dos arquivos, substituindo arquivos no destino sem solicitar confirmação.

Resultado: Todos os arquivos e subdiretórios do diretório de origem serão copiados para o diretório de destino, que inclui a data no nome.

3. Registrando a Conclusão do Backup em um Arquivo de Log

- **\$logMessage**: Esta variável armazena uma mensagem de log, que inclui a informação "Backup concluído em " seguida pela data e hora atual, formatada como **"dd/MM/yyyy HH:mm"**.
- **\$logPath**: Esta variável armazena o caminho e nome do arquivo de log onde a mensagem será gravada.
- **Add-Content**: Este comando adiciona o conteúdo especificado (neste caso, a mensagem de log) ao final de um arquivo. Se o arquivo não existir, ele será criado.

Resultado: Após a conclusão da cópia dos arquivos, o script registra no arquivo de log a data e hora em que o backup foi concluído.

Funcionamento Geral do Script

- **Propósito:** O script é projetado para criar uma cópia de segurança dos arquivos de um diretório de origem para um diretório de destino, onde o diretório de destino inclui a data do backup no seu nome. Além disso, o script registra no log a data e hora em que o backup foi concluído.
- **Automação:** Este script pode ser agendado para ser executado automaticamente em intervalos regulares (por exemplo, diariamente) para garantir que backups regulares sejam feitos sem intervenção manual.
- **Flexibilidade:** Como o destino do backup é nomeado com a data, cada execução do script cria um novo diretório de backup, evitando sobrescrever backups anteriores e mantendo um histórico das cópias de segurança.

6.2 Script para Limpeza de Arquivos Temporários

```
# Define o diretório temporário do sistema
$temp_dir = $env:TEMP
# Exclui todos os arquivos no diretório temporário
Get-ChildItem -Path $temp_dir -Recurse -Force | Remove-Item -Force
# Remove o próprio diretório temporário (e seu conteúdo se ainda houver algum)
Remove-Item -Path $temp_dir -Recurse -Force
# Registra a limpeza no log com a data e hora atuais
```

```
$logMessage = "Diretório de arquivos temporários limpo em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss")
$logPath = "C:\Logs\temp_cleanup_log.txt"
Add-Content -Path $logPath -Value $logMessage
```

Explicação do Script

1. Definindo o Diretório Temporário:
 - **\$env:TEMP**: Obtém o caminho do diretório temporário do sistema, que é armazenado na variável de ambiente TEMP.
2. Excluindo Todos os Arquivos no Diretório Temporário:
 - **Get-ChildItem -Path \$temp_dir -Recurse -Force**: Lista todos os arquivos e subdiretórios no diretório temporário recursivamente, incluindo arquivos ocultos e de sistema.
 - **Remove-Item -Force**: Exclui os arquivos e diretórios listados sem solicitar confirmação.
3. Removendo o Diretório Temporário:
 - **Remove-Item -Path \$temp_dir -Recurse -Force**: Remove o diretório temporário e qualquer conteúdo restante que não tenha sido excluído na etapa anterior.
4. Registrando a Limpeza no Log:
 - **\$logMessage**: Cria uma mensagem de log contendo a informação de que a limpeza foi concluída, juntamente com a data e hora atuais.
 - **Add-Content -Path \$logPath -Value \$logMessage**: Adiciona a mensagem de log ao arquivo **temp_cleanup_log.txt** no diretório **C:\Logs**. Se o arquivo não existir, ele será criado.

Funcionamento Geral do Script

- **Objetivo**: Este script limpa o diretório temporário do sistema excluindo todos os arquivos e pastas dentro dele e, em seguida, registra a operação em um arquivo de log.
- **Automatização**: O script pode ser executado manualmente ou agendado para execução periódica para manter o diretório temporário limpo, o que pode ajudar a liberar espaço em disco e melhorar o desempenho do sistema.
- **Registro de Atividade**: Mantém um log das operações de limpeza, incluindo a data e hora em que foram realizadas, o que é útil para monitorar e auditar a manutenção do sistema.

6.3 Script para Monitoramento de Logs do Event Viewer

```
# Define o caminho do arquivo de log
$logfile = "C:\Logs\event_log.txt"
# Extrai os últimos 10 eventos do log de Aplicações e salva no arquivo de log
wevtutil qe Application /rd:true /f:text /c:10 > $logfile
# Adiciona uma linha no arquivo de log com a data e hora da monitorização
$logMessage = "Logs monitorados em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss")
Add-Content -Path $logfile -Value $logMessage
```

Explicação do Script

- **\$logfile = "C:\Logs\event_log.txt"**: Define a variável \$logfile com o caminho onde o arquivo de log será salvo.
- **wevtutil qe Application /rd /f /c:10 > \$logfile** - Usa o comando wevtutil para extrair os últimos 10 eventos do log de Aplicações do Event Viewer, formatados como texto, e salva a saída no arquivo de log especificado por \$logfile.

- `$logMessage = "Logs monitorados em " + (Get-Date -Format "dd/MM/yyyy HH:mm"):`
Cria uma mensagem de log contendo a data e hora atuais.
- `Add-Content -Path $logfile -Value $logMessage:` Adiciona a mensagem de log ao final do arquivo, registrando quando os eventos foram monitorados.

6.4 Backup de Documentos Pessoais

```
# Define o caminho da origem
$source = "C:\Users\[pasta_do_usuario]\Documentos"
# Cria a pasta de destino com a data formatada
$destination = "D:\Backup\Documentos\backup_" + (Get-Date -Format "dd-MM-yyyy")
# Verifica se a pasta de destino existe, se não, cria a pasta
if (-not (Test-Path -Path $destination)) {
    New-Item -ItemType Directory -Path $destination
}
# Copia os arquivos da origem para o destino, incluindo subpastas e arquivos ocultos
Copy-Item -Path $source\* -Destination $destination -Recurse -Force
# Registra a data e a hora no log
$logEntry = "Backup de Documentos concluído em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss") + "`r`n"
Add-Content -Path "D:\Backup\Documentos\backup_log.txt" -Value $logEntry
```

Explicação dos comandos.

- **Formatação de data:** A formatação da data no PowerShell é feita usando `Get-Date -Format "formato"`.
- **Test-Path e New-Item:** Esses comandos verificam se o diretório de destino já existe e, se não, criam o diretório.
- **Copy-Item:** É o comando usado para copiar arquivos e diretórios no PowerShell. A opção `-Recurse` garante que todas as subpastas e arquivos sejam copiados, e a opção `-Force` permite substituir arquivos existentes.
- **Add-Content:** Este comando adiciona a entrada de log ao arquivo especificado, criando-o se não existir.

6.5 Backup de Projetos de Desenvolvimento

```
# Define o caminho da origem
$source = "C:\Desenvolvimento\Projetos"
# Cria a pasta de destino com a data formatada
$destination = "\\Servidor\Backups\Projetos\backup_" + (Get-Date -Format "dd-MM-yyyy")
# Verifica se a pasta de destino existe, se não, cria a pasta
if (-not (Test-Path -Path $destination)) {
    New-Item -ItemType Directory -Path $destination
}
# Copia os arquivos da origem para o destino, incluindo subpastas e arquivos ocultos
Copy-Item -Path $source\* -Destination $destination -Recurse -Force
# Registra a data e a hora no log
$logEntry = "Backup de Projetos concluído em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss") + "`r`n"
Add-Content -Path "\\Servidor\Backups\Projetos\backup_log.txt" -Value $logEntry
```

Explicação dos comandos

- **Variáveis:** `$source` e `$destination` armazenam os caminhos de origem e destino.
- **Formatação da data:** Usamos `Get-Date -Format "dd-MM-yyyy"` para formatar a data na criação da pasta de backup.
- **Test-Path e New-Item:** Verifica se o diretório de destino existe e cria-o se necessário.
- **Copy-Item:** Este comando copia todos os arquivos e subpastas do diretório de origem para o diretório de destino, incluindo arquivos ocultos.
- **Add-Content:** Adiciona uma linha ao arquivo de log com a data e hora da conclusão do backup.

Automatizar a Limpeza de Arquivos Temporários

Exemplo 1: Manutenção de Sistema em um PC Pessoal

```
#Limpar o diretório Temp da pasta do usuário atual
# Define o diretório temporário
$temp_dir = $env:TEMP
# Exclui todos os arquivos no diretório temporário
Get-ChildItem -Path $temp_dir\* -Recurse | Remove-Item -Force
# Remove o diretório temporário e todas as subpastas
Remove-Item -Path $temp_dir -Recurse -Force
# Registra a data e a hora no log
$logEntry = "Diretório de arquivos temporários limpo em " + (Get-Date -Format
"dd/MM/yyyy HH:mm:ss") + "`r`n"
Add-Content -Path "C:\Logs\temp_cleanup_log.txt" -Value $logEntry
```

Explicação do código.

- **Variável \$temp_dir:** O PowerShell usa \$env:TEMP para acessar a variável de ambiente TEMP.
- **Remoção de arquivos:** Get-ChildItem -Path \$temp_dir* -Recurse | Remove-Item -Force lista e remove todos os arquivos no diretório temporário, incluindo os arquivos em subpastas.
- **Remoção de diretório:** Remove-Item -Path \$temp_dir -Recurse -Force remove o diretório temporário e todas as subpastas.
- **Log:** Add-Content adiciona uma linha ao arquivo de log com a data e hora da limpeza.

Exemplo 2: Manutenção de Servidor de Arquivos

```
#Semelhante ao anterior porém para ser executado em rede, para servidores
#Define o diretório temporário
$temp_dir = $env:TEMP
# Exclui todos os arquivos no diretório temporário
Get-ChildItem -Path $temp_dir\* -Recurse | Remove-Item -Force
# Remove o conteúdo do diretório temporário
Get-ChildItem -Path $temp_dir\* -Recurse | Remove-Item -Recurse -Force
# Registra a data e a hora no log
$logEntry = "Diretório de arquivos temporários limpo em " + (Get-Date -Format
"dd/MM/yyyy HH:mm:ss") + "`r`n"
Add-Content -Path "\\Servidor\Logs\temp_cleanup_log.txt" -Value $logEntry
```

- **Definição do diretório temporário:**

A variável \$temp_dir é usada para armazenar o caminho para o diretório temporário (\$env:TEMP).

- **Remoção de arquivos:**

Get-ChildItem -Path \$temp_dir* -Recurse | Remove-Item -Force lista todos os arquivos e subpastas no diretório temporário e os remove. A opção -Recurse garante que todos os arquivos em subpastas também sejam removidos.

- **Remoção de subpastas:**

A remoção é feita de forma recursiva com Remove-Item -Recurse -Force, que remove os arquivos e pastas dentro do diretório temporário, mas não tenta remover o próprio diretório, pois ele é um diretório do sistema que será recriado automaticamente.

- **Registro em log:**

O script adiciona uma entrada ao arquivo de log \\Servidor\Logs\temp_cleanup_log.txt, registrando a data e hora da limpeza.

- **Exemplo 1: Monitoramento de Desempenho de um Servidor ou Desktop**

```
# Define o caminho do arquivo de log
$logfile = "C:\Logs\server_usage_log.txt"
# Função para formatar bytes em KB, MB, GB
function Format-Size ($size) {
    if ($size -ge 1GB) {
        return "{0:N2} GB" -f ($size / 1GB)
    }
    elseif ($size -ge 1MB) {
        return "{0:N2} MB" -f ($size / 1MB)
    }
    elseif ($size -ge 1KB) {
        return "{0:N2} KB" -f ($size / 1KB)
    }
    else {
        return "$size Bytes"
    }
}
# Função para formatar o tempo de CPU em horas, minutos e segundos
function Format-CPUTime ($cpuTime) {
    $timespan = [TimeSpan]::FromSeconds($cpuTime)
    return $timespan.ToString("hh:mm:ss")
}
# Loop infinito
while ($true) {
    # Adiciona a data e a hora ao log e exibe na tela
    $timestamp = Get-Date -Format "dd/MM/yyyy HH:mm:ss"
    Write-Host $timestamp
    $timestamp | Out-File -FilePath $logfile -Append

    # Obtém os 10 processos que mais consomem CPU e formata WS, PM e CPU
    $processInfo = Get-Process | Sort-Object CPU -Descending | Select-Object -First 10 |
    ForEach-Object {
        [PSCustomObject]@{
            Name = $_.Name
            CPU = Format-CPUTime $_.CPU
            WS = Format-Size $_.WorkingSet
            PM = Format-Size $_.PagedMemorySize
        }
    } | Format-Table -AutoSize | Out-String -Width 512
    # Exibe na tela e registra no log
    Write-Host $processInfo
    $processInfo | Out-File -FilePath $logfile -Append
    # Adiciona uma linha em branco ao log e exibe na tela
    "r`n" | Out-File -FilePath $logfile -Append
    Write-Host ""
    # Aguarda 300 segundos (5 minutos)
    Start-Sleep -Seconds 300
}
```

- **Exemplo 1: Manutenção de Disco Rígido em um PC Pessoal**

```
# Define o caminho do arquivo de log
$logfile = "C:\Logs\defrag_log.txt"
# Executa o comando de desfragmentação no drive C: e registra o output no log
```

```
Start-Process -FilePath "defrag.exe" -ArgumentList "C:", "/U", "/V" -NoNewWindow -
RedirectStandardOutput $logfile -Wait
```

Sincronização de Diretórios com Robocopy

- **Exemplo 1: Sincronização de Diretórios Entre Computadores**

```
# Define os caminhos de origem e destino
$source = "C:\Trabalho"
$destination = "\\OfficePC\Trabalho" # Deve definir para qual destino será sincronizado o
conteúdo da pasta trabalho
# Define o caminho do arquivo de log
$logfile = "C:\Logs\robocopy_log.txt"
# Executa o comando Robocopy com os parâmetros especificados
Start-Process -FilePath "robocopy.exe" -ArgumentList "$source", "$destination", "/MIR",
"/R:5", "/W:10", "/LOG:$logfile" -NoNewWindow -Wait
```

Exemplo 2: Backup Incremental de Arquivos

```
# Define os caminhos de origem e destino
$source = "C:\Projetos"
$destination = "D:\Backup\Projetos"

# Define o caminho do arquivo de log
$logfile = "C:\Logs\backup_incremental_log.txt"

# Executa o comando Robocopy com os parâmetros especificados
Start-Process -FilePath "robocopy.exe" -ArgumentList "$source", "$destination", "/MIR",
"/R:5", "/W:10", "/LOG:$logfile" -NoNewWindow -Wait
```

Explicação do script

- o destino (\$destination) é ajustado para um disco local, D:\Backup\Projetos, em vez de um caminho de rede.
- **Outras partes do script:**
- As variáveis \$source e \$logfile continuam definindo os caminhos de origem e de log, respectivamente.
- O comando Start-Process chama robocopy.exe com os mesmos parâmetros:
 - /MIR: Mantém o espelho da origem no destino.
 - /R:5: Tenta repetir a cópia até 5 vezes em caso de falha.
 - /W:10: Aguarda 10 segundos entre tentativas.
 - /LOG:\$logfile: Registra a saída em um arquivo de log.
- -NoNewWindow e -Wait garantem que o processo seja executado na mesma janela e o script espere pela conclusão.

Criação de Ponto de Restauração do Sistema

- **Exemplo 1: Criação Automática de Pontos de Restauração Antes de Instalar Atualizações**

```
# Cria um ponto de restauração do sistema com a descrição especificada
Checkpoint-Computer -Description 'Antes da Atualização' -RestorePointType
'MODIFY_SETTINGS'
# Registra a criação do ponto de restauração no arquivo de log
$timestamp = Get-Date -Format "dd/MM/yyyy HH:mm:ss"
$logMessage = "Ponto de restauração criado antes da atualização em $timestamp"
$logMessage | Out-File -FilePath "C:\Logs\restore_point_log.txt" -Append
```


Script para Backup Compactador de Fotos

```
# Define os caminhos de origem e destino
$source = "C:\Fotos"
$destination = "D:\Backup\Fotos\backup_" + (Get-Date -Format "dd-MM-yyyy") + ".zip"
# Compacta os arquivos do diretório de origem para o arquivo zip de destino
Compress-Archive -Path "$source\*" -DestinationPath $destination
# Registra a operação no arquivo de log com a data e hora atuais
$logMessage = "Fotos compactadas em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss")
$logPath = "D:\Backup\Fotos\backup_log.txt"
Add-Content -Path $logPath -Value $logMessage
```

Explicação do Script.

1. Definindo os Caminhos de Origem e Destino:

- \$source = "C:\Fotos": Define o diretório de origem onde as fotos estão localizadas.
- \$destination = "D:\Backup\Fotos\backup_" + (Get-Date -Format "dd-MM-yyyy") + ".zip": Define o caminho de destino para o arquivo compactado (.zip), com a data atual incluída no nome do arquivo.

2. Compactação dos Arquivos:

- Compress-Archive -Path "\$source*" -DestinationPath \$destination: Usa o comando Compress-Archive para compactar todos os arquivos e subdiretórios do diretório de origem em um arquivo .zip no diretório de destino especificado.

3. Registro da Operação no Log:

- \$logMessage = "Fotos compactadas em " + (Get-Date -Format "dd/MM/yyyy HH:mm:ss"): Cria uma mensagem de log que inclui a data e hora em que a compactação foi concluída.
- \$logPath = "D:\Backup\Fotos\backup_log.txt": Define o caminho onde o arquivo de log será salvo.
- Add-Content -Path \$logPath -Value \$logMessage: Adiciona a mensagem de log ao arquivo backup_log.txt.

Verificação e Correção de Erros no Disco

• Exemplo 1: Manutenção Preventiva em um PC Pessoal

```
# Verifica se o script está sendo executado como administrador
If (-Not ([Security.Principal.WindowsPrincipal]
[Security.Principal.WindowsIdentity]::GetCurrent()).IsInRole([Security.Principal.WindowsBuiltInRole] "Administrator"))
{
    Write-Host "Este script precisa ser executado como administrador. Por favor, execute o
PowerShell com privilégios elevados." -ForegroundColor Red
    Exit
}

# Executa o comando CHKDSK no drive C: e registra a saída no log
Start-Process -FilePath "chkdsk.exe" -ArgumentList "C: /F /R" -NoNewWindow -
RedirectStandardOutput "C:\Logs\chkdsk_log.txt"
```

Exportação de Políticas de Segurança para Backup

Este script deve ser salvo em um arquivo com extensão ps1 e executado como administrador do sistema.

```
# Define o caminho do diretório de backup e do arquivo de configuração
$backupDir = "C:\Backup"
$cfgFile = "$backupDir\secpol.cfg"
# Verifica se o diretório de backup existe; se não existir, cria o diretório
if (-Not (Test-Path -Path $backupDir)) {
    try {
        New-Item -Path $backupDir -ItemType Directory -ErrorAction Stop | Out-Null
        Write-Host "Diretório de backup criado: $backupDir"
    } catch {
        Write-Host "Erro ao criar o diretório de backup: $_" -ForegroundColor Red
        Exit
    }
}
# Exporta as configurações de segurança para o arquivo .cfg
try {
    Start-Process -FilePath "secedit.exe" -ArgumentList "/export /cfg $cfgFile" -NoNewWindow -
Wait -ErrorAction Stop
    if (Test-Path -Path $cfgFile) {
        Write-Host "Configurações de segurança exportadas com sucesso para: $cfgFile"
    } else {
        Write-Host "Falha na exportação das configurações de segurança. O arquivo não foi
criado." -ForegroundColor Red
        Exit
    }
} catch {
    Write-Host "Erro ao exportar as configurações de segurança: $_" -ForegroundColor Red
    Exit
}
# Registra a exportação das configurações de segurança no log com a data e hora atuais
$logMessage = "Configurações de segurança exportadas em " + (Get-Date -Format
"dd/MM/yyyy HH:mm:ss")
$logPath = "C:\Logs\secpol_export_log.txt"
Add-Content -Path $logPath -Value $logMessage
Write-Host "Operação registrada no log: $logPath"
```

Backup do registro do Windows

```
# Define o caminho para o backup do registro e o log
$regBackupFile = "C:\Backup\registro_backup.reg"
$logFile = "C:\Logs\registry_backup_log.txt"
# Verifica e cria o diretório de backup se necessário
if (-Not (Test-Path -Path "C:\Backup")) {
    New-Item -Path "C:\Backup" -ItemType Directory | Out-Null
}
# Faz o backup de todo o registro
reg export HKLM $regBackupFile /y
# Registra a operação no log
$logMessage = "Backup geral do registro realizado em " + (Get-Date -Format "dd/MM/yyyy
HH:mm:ss")
Add-Content -Path $logFile -Value $logMessage
```

Restauração do Backup do registro do Windows

```
# Define o caminho para o arquivo de backup do registro
$regBackupFile = "C:\Backup\registro_backup.reg"
$logFile = "C:\Logs\registry_restore_log.txt"
```

```
# Verifica se o arquivo de backup existe
if (Test-Path -Path $regBackupFile) {
    try {
        # Restaura o registro a partir do backup
        reg import $regBackupFile

        # Registra a operação no log
        $logMessage = "Restauração do registro realizada em " + (Get-Date -Format "dd/MM/yyyy
HH:mm:ss")
        Add-Content -Path $logFile -Value $logMessage
        Write-Host "Restauração do registro concluída com sucesso."
    } catch {
        Write-Host "Erro ao restaurar o registro: $_" -ForegroundColor Red
    }
} else {
    Write-Host "Arquivo de backup do registro não encontrado: $regBackupFile" -
ForegroundColor Red
}
```

Recursos Adicionais

- **Documentação Oficial da Microsoft:** [Documentação do Prompt de Comando](#)
- **Tutoriais Online:** Existem vários tutoriais em vídeo no YouTube que demonstram o uso do Prompt de Comando em cenários práticos.

Este guia deve servir como uma referência útil para aprender e dominar o uso do Prompt de Comando no Windows. Pratique esses comandos regularmente para aumentar sua familiaridade e proficiência.