**Powershell Parte 1 – Começando Com A Linguagem De Scpripting Moderna**

1. **Aula 1 – Apresentando o PowerShell:**
   1. Conhecendo o problema do cliente:
      1. Mstsc: Ao rodar no executar win+R, abre uma conexão de área remota.
      2. Podemos ver no gerenciador de tarefas o que está consumindo muito da máquina, clicar com o botão direito e ir para detalhes e a partir dos detalhes, clicar com botão direito e ir para serviços, assim descobrimos quais serviços estão utilizando aquela aplicação que está consumindo muito do nosso sistema.
      3. O problema do nosso cliente é exatamente esse, temos um serviço que está usando 90/100% da capacidade da cpu dos nossos servidores e precisamos reiniciar eles para que voltem ao normal, mas, conectar remotamente em um por um e fazer manualmente nos 10 servidores é muito trabalhoso, por isso vamos escrever um script que faça esse trabalho.
   2. Tentando solucionar com o antigo CMD:
      1. Criamos um script que identifica se um serviço está rodando ou não e se estiver faz reiniciar, se não estiver retorna que não está.
      2. O CMD não possui uma convenção de nomes definida. Ao contrário do PowerShell, o CMD não possui uma forte convenção de nomes.
      3. O CMD só trabalha com texto! Ou seja, para obtermos as informações que nos interessam em um script, muitas vezes, precisamos lidar com busca de palavras chaves no retorno dos comandos. Diferente do PowerShell, o CMD não é orientado a objetos.
      4. Muitas vezes é necessário conhecimento específico sobre os comandos que executamos, pois, cada um recebe argumentos e funciona de uma maneira diferente. O CMD está disponível até nas versões mais recentes do windows, como o Windows 10 ou o Windows Server 2016.
   3. Resolvendo o problema com PowerShell:
      1. Diferente do cmd, no powershell é tudo bem mais intuitivo, por exemplo, para obter os serviços vc digita exatamente isso Get-Service.
      2. Quando der enter o PS te devolve todos os serviços, parados e rodando, da sua máquina.
      3. Ao contrário do cmd que tínhamos que digitar sc query e afins, sem nem sequer sabermos para que serve ou que exatamente significa esse comando.
      4. Tudo no PS tem autocomplete, ou seja, se não souber o nome completo do comando, basta digitar uma parte dele e ir apertando tab até encontrar o que precisa.
      5. $variável = : Cria uma variável com seja lá o que você quiser. Pode inclusive obter um serviço e colocar dentro de uma variável:

$servico = Get-Service -Name "Themes"



* + - 1. Ao colocar um ‘.’ Depois da variável e ir dando tab podemos ver todas as funções que podemos utilizar com ela, como por exemplo:



* + 1. Ao invés de ficar reescrevendo todo o código redundantemente como no cmd, podemos fazer simplesmente assim:



* + 1. Já que o PS é orientado a objeto, tudo fica mais fácil e simples.

1. **Aula 2 – Usando Comandos Legados e Aprendendo Sem o Google:**
   1. Conhecendo o CommandPrecedence:
      1. Como questão de segurança, o PS faz com que todos os scipts do cmd escritos não rodem nele, mas, se realmente quiser rodar um script de cmd, precisa colocar ‘.\’ antes do .bat que quer executar, dessa forma ele irá rodar normalmente.
      2. Esse é o famoso commandPrecedence.
   2. Quando o CommandPrecedence não é necessário e o comando GetMember:
      1. Diferentemente do cmd, para podermos ver o nosso path, precisamos dar um echo na variável $env:path. Não só para o path, mas qualquer outra variável de ambiente que você queira. Basta substituir o path por ela, ou ir dando tab até encontrar.
      2. Para saber qual é o tipo de uma variável podemos colocar $env:variável.GetType().



* + 1. O path é sempre muito confuso principalmente quando está com muitos caminhos, mas, como podemos ver, ele é uma string portanto os métodos de string vão funcionar com ele também, e podemos utilizá-los para ter uma melhor exibição dele.
    2. Para procurar qualquer documentação na Microsoft sobre métodos ou suas linguagens de programação digite msdn e o que está procurando, como no caso de strings.
    3. O primeiro link nos leva pra cá: <https://docs.microsoft.com/en-us/dotnet/api/system.string?view=net-6.0>.
    4. Para deixar nosso path mais visível podemos usar o método split() de strings, passando como parâmetro o ‘;’ para fazer essa separação:



* + 1. Nem sempre teremos acesso a internet para poder pesquisar um método específico para trabalhar com nossa variável. Por conta disso podemos utilizar o comando Get-Member, que nos retorna uma lista de tudo o que podemos utilizar com nossa variável e suas descrições:



* 1. Apelidos para comandos antigos:
     1. Muitos comandos antigos do cmd, por convenção, ganharam apelidos devido ao seu uso muito recorrente, tal como o cd e o dir, ou seja, ao invés de digitar o que você quer fazer, como o get-disk, por exemplo, nós podemos continuar usando cd e dir.
     2. Para ver quais são os apelidos dados para os comandos nós utilizamos o get-alias, ou seja, obter apelidos:

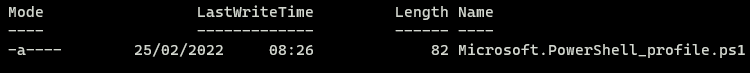


* + 1. Para procurar algum específico podemos utilizar o argumento -Name nome:

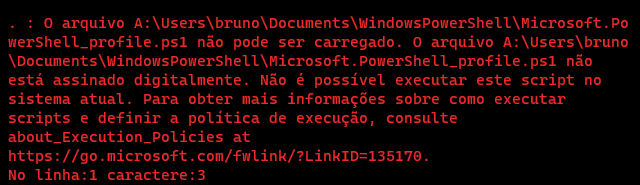


* 1. Descobrindo comandos sem o Google e criando novos apelidos:
     1. Get-command: Lista todos os comandos existentes no PS.
        1. Para procurar um específico podemos ver os argumentos que podemos passar para ele e temos -Name, -Verbose, -Verb dentre outros. Como queremos achar o rename, podemos utilizar o -Verb, uma vez renomear é um verbo.
        2. Entretanto, colocamos o rename entre ‘\*’, para que ele sirva como coringa e traga tudo o que achar que possua esse verbo.
     2. Isso só é possível pois a nova nomenclatura de comandos no PS é composta por verb-noun, ou seja, um verbo e um noum.
     3. Para obter ajuda sobre algum comando ou qualquer coisa no PS, nós utilizamos um comando que pede exatamente isso: Get-Help -Name comando e ele nos devolve uma lista e texto com a ajuda do comando pedido.
     4. Para voltar o tab caso tenha encontrado o que procura, utilizamos shift+tab.
     5. -WhatIf: é um switch argument, ou seja, não precisa de valor nenhum após ter sido colocado. Esse argumento não executará o comando que colocamos, mas sim nos dirá o que aconteceria se o executarmos para evitar que utilizemos comandos que não conhecemos direito e façamos alguma coisa de errado.
     6. Para setar um novo alias utilizamos o comando new-alias -Name “apelido\_do\_comando” comando-em-si: New-Alias -name "ren" Rename-Item

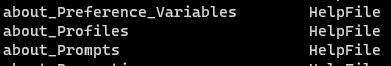
1. **Aula 3 – Políticas de Execução e Profiles:**
   1. Usando PROFILES no PowerShell:
      1. Profiles são scripts que criamos e que rodam automaticamente sempre que iniciamos o PS.
      2. Podemos ver nossos profiles a partir da variável $PROFILE que, além de retornar todo o caminho até a pasta do PS que estão nossos profiles, exibe o nome e a extensão deles.
      3. A extensão de scripts que rodam no PS é .ps1, diferentemente do cmd que é .bat.
      4. Se não tiver nada dentro da pasta de profiles, ou seja, nenhum profile definido podemos criar a partir do comando New-Item $PROFILE.
      5. Ni ou new-item, cria um arquivo ou diretório a depender do diretório em que é executado. Descrição do comando na ajuda do PS:
         1. “The `New-Item` cmdlet creates a new item and sets its value. The types of items that can be created depend on the location of the item. For example, in the file system, `New-Item` creates files and folders. In the registry, `New-Item` creates registry keys and entries.
         2. `New-Item` can also set the value of the items that it creates. For example, when it creates a new file, `New-Item` can add initial content to the file.”
      6. Ou seja, ao executar o ni dentro de profile, ele automaticamente já cria um arquivo .ps1 com o nome correto:



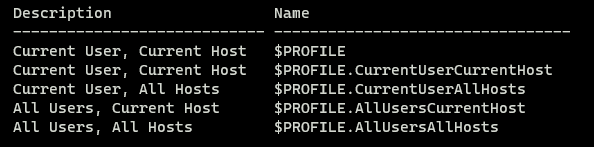
* + 1. A partir daí podemos abrir o arquivo com um editor de texto, sendo notepad ou o vim no próprio terminal, e escrever nosso script.
    2. Infelizmente ele pode dar um pequeno erro de execução quando reiniciarmos o PS para testar se nosso profile funcionou:



* 1. Entendendo e alterando as políticas de execução:
     1. Para obter ajuda sobre a política de execução basta dar um get-help about\_Exec...
     2. Basicamente o PS vem como padrão restrito por segurança, o que impede que possamos executar qualquer script de powershell nele.
     3. Existem vários outros métodos que podemos deixar o nosso OS:
        1. AllSigned:
           1. Scripts can run.
           2. Requires that all scripts and configuration files be signed by a trusted publisher, including scripts that you write on the local computer.
           3. Prompts you before running scripts from publishers that you haven't yet classified as trusted or untrusted.
           4. Risks running signed, but malicious, scripts.
        2. Bypass:
           1. Nothing is blocked and there are no warnings or prompts.
           2. This execution policy is designed for configurations in which a PowerShell script is built in to a larger application or for configurations in which PowerShell is the foundation for a program that has its own security model.
        3. Default:
           1. Sets the default execution policy.
           2. RESTRICTED for Windows clients.
           3. REMOTESIGNED for Windows servers.
        4. RemoteSigned:
           1. The default execution policy for Windows server computers.
           2. Scripts can run.
           3. Requires a digital signature from a trusted publisher on scripts and configuration files that are downloaded from the internet which includes email and instant messaging programs.
           4. Doesn't require digital signatures on scripts that are written on the local computer and not downloaded from the internet.
           5. Runs scripts that are downloaded from the internet and not signed, if the scripts are unblocked, such as by using the Unblock-File cmdlet.
           6. Risks running unsigned scripts from sources other than the internet and signed scripts that could be malicious.
        5. Restricted:
           1. The default execution policy for Windows client computers.
           2. Permits individual commands, but does not allow scripts.
           3. Prevents running of all script files, including formatting and configuration files (.ps1xml), module script files (.psm1), and PowerShell profiles (.ps1).
        6. Undefined:
           1. There is no execution policy set in the current scope.
           2. If the execution policy in all scopes is UNDEFINED, the effective execution policy is RESTRICTED for Windows clients and REMOTESIGNED for Windows Server.
        7. Unrestricted:
           1. Unsigned scripts can run. There is a risk of running malicious scripts.
           2. Warns the user before running scripts and configuration files that are not from the local intranet zone.
           3. [!NOTE] On systems that do not distinguish Universal Naming Convention (UNC) paths from internet paths, scripts that are identified by a UNC path might not be permitted to run with the REMOTESIGNED execution policy.
     4. Get-ExecutionPolicy: Para saber qual é a política de execução atual.
     5. Set-ExecutionPolicy política\_de\_execução: Para setar uma nova política das disponíveis acima.
     6. A minha política de execução estava como allSigned pois precisamos deixar nessa política para poder instalar o chocolatey, mas mesmo assim, mesmo não estando no restrito, ele não me permitiu rodar o profile do PS que criamos.
        1. Ele só me permitiu rodar o profile quando setei como Unrestricted.
        2. Só podemos fazer a alteração de política em modo de administrador.
  2. Entendendo o que são Hosts e criando outros Profiles:
     1. Host é tudo o que hospeda algo, no nosso caso do PS o host é o nosso perfil de login, o padrão, mas se tivéssemos outros perfis na mesma máquina, eles seriam hosts diferentes.
     2. Todas as alterações que fazemos com o script do profile só são válidas para o nosso host, ou seja, o perfil na máquina que se refere a nós, porém, quando outra pessoa que possui outro perfil na máquina for acessar o PS e tentar usar os alias que nós criamos automaticamente pelo script do nosso perfil, ela não conseguirá, justamente pq o script é exclusivo do nosso perfil.
     3. Por conta disso podemos utilizar as propriedades existentes do perfil para criar algo que sirva para todos os hosts e todos os perfis.
     4. Para ver quais são as propriedades existentes da variável $PROFILE podes dar um get-help em about\_\*, nos retornando todos os arquivos de ajuda que possuem o about no nome. A partir daí podemos procurar o que se refere ao perfil:

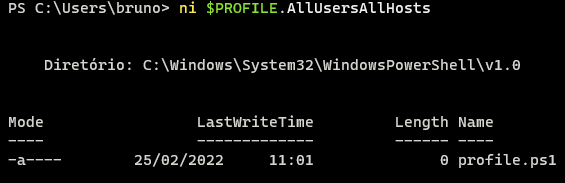


* + 1. E obter ajuda desse documento com get-help about\_profiles.

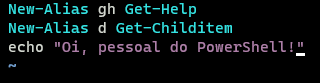


* + 1. Dentro da ajuda nós encontramos essa tabelinha que nos diz as exatas propriedades que estávamos procurando. Queremos fazer uma alteração em todos os usuários e todos os hosts para que esses alias sejam configurados em todos os lugares para que a equipe toda possa utilizar eles, e não só nós.
    2. Para fazer isso, basta seguir o mesmo exemplo de criação de um script de profile para o usuário atual, mas sempre colocando o .AllUsersAllHost depois de $PROFILE:





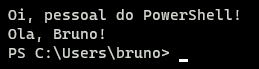
vim $PROFILE.AllUsersAllHosts



vim $PROFILE



* + - 1. Lembrando que ni é alias de new-item.
      2. Utilizei o vim para fazer a edição do profile.ps1 criado pelo ni, mas poderia facilmente ser o notepad. Utilizei chocolatey com choco install vim para instalar o editor de texto de terminal.
      3. O get-childitem é o comando completo dos alias dir e do ls, criamos um novo alias chamado d.
      4. O segundo vim foi para removermos os códigos do new-alias que fizemos somente no nosso perfil, pois se fizéssemos o script para todos os usuários e hosts e ainda mantivéssemos o nosso pessoal, daria um erro de execução na hora de abrir o PS, dizendo que ao alias já haviam sido criados. Mantivemos somente o nosso olá nesse arquivo. Esse olá só aparecerá para nós pois está no script do profile pessoal.
    1. Ao abrir PS:

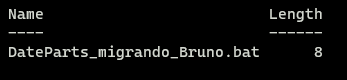


* + - 1. O olá pessoal aparece para todos e o ola bruno somente para mim.
  1. O que aprendemos:
     1. O que são políticas de execução;
     2. Alterar políticas de execução;
     3. Criar um perfil no PowerShell;
     4. O que são Hosts;
     5. Criar um profile em todos os hosts e perfis.

1. **Aula 4 – Avançando na Sintaxe do PowerShell:**
   1. Filtrando e formatando uma coleção:
      1. Gci/dir/ls -Recursive: Lista tudo que existe da pasta em que o comando foi utilizado para frente ou seja, dentro dessa pasta e de todas as subpastas.
      2. Podemos guardar o resultado retornado em uma variável $variável. O que nós recebemos como resultado é um Array/lista.
      3. A partir disso podemos utilizar diversas propriedades com essa variável, uma vez que recebemos um Array. Podemos verificar o tamanho com Length.
      4. Entretanto essa propriedade mostra quantos arquivos E diretórios eu tenho, e não somente os arquivos. Para ver todos os arquivos podemos passar outro argumento na hora do gci: -File. Ele é um switch argumment que irá listar somente os arquivos, ignorando os diretórios.
      5. Ao usar o length na nossa variável agr, o resultado retornado é a quantidade de arquivos apenas.
      6. gci -Recurse -File | Select-Object Name | Where-Object {$\_ -like "\*\_migrando\_\*"}
         1. Esse é um Código que fizemos para retornar somente os arquivos que possuíam o nome migrando.
         2. Basicamente o que ele está dizendo é para listar todos os arquivos dessa pasta pra frente, o resultado disso seleciona somente o nome dos objetos, depois de selecionado exibe somente os objetos que possuem a palavra \*\_migrando\_\* no nome, ou seja, não importa o que vem ante ou depois desde que tenha a palavra, isso que significa os ‘\*’.
         3. A variável $\_ é um iterador, ou seja, ela diz para verificar se existe essa palavra em cada um dos objetos selecionados anteriormente.
         4. O -like indica que é para procurar a palavra parecida com a que está em “”.
         5. Os {} são blocos de script que se repetem.
         6. Resultado Final:



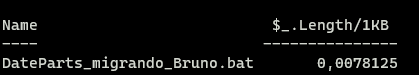
* + 1. Com esse código acima não podemos fazer mais nada com o que é retornado, como ver o tamanho de cada item, por exemplo.
       1. Para isso podemos inverter a ordem do Where e do select object:
       2. gci -Recurse -File | Where-Object Name -like "\*\_migrando\_\*" | Select-Object Name, Length
       3. Faz exatamente a mesma coisa apesar de devolver de forma diferente:



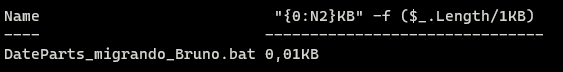
* 1. Aplicando conversões de bytes:
     1. A diferença do primeiro para o segundo códigos acima é que, no primeiro nós primeiro fizemos a formatação com o select e depois filtramos utilizando o Where, já no segundo fizemos o contrário, primeiro filtramos e depois fizemos a formatação.
     2. Podemos fazer conversões e formatações de strings no ps:
        1. "{0:N2}GB = {1:N2}MB" -f (5445644654/1GB), (5445644654/1MB)



* + - 1. -f: faz a formatação do que vem a seguir na str indicada antes.
      2. O 0 e 1 antes do :N2 é index de cada item que vem depois do -f.
      3. O resto é igual python.
  1. Usando ScriptBlocks com o SelectObject:
     1. Para fazer o nosso length ser divido por 1KB para conseguirmos ler o valor melhor, precisamos fazer a iteração de cada item encontrado pelo filtro e dividir a length dele por 1KB. Basicamente pegamos o segundo código de 2 tópicos atrás e colocamos o LENGTH dentro de {} depois do $\_:
        1. gci -Recurse -File | Where-Object Name -like "\*\_migrando\_\*" | Select-Object Name, { $\_.Length/1KB }:



* + 1. Ainda não está legível do jeito que gostaríamos, com 2 casas numéricas e escrito KB depois. Para concertar utilizamos o format que aprendemos no tópico anterios:
       1. gci -Recurse -File | Where-Object Name -like "\*\_migrando\_\*" | Select-Object Name, { "{0:N2}KB" -f ($\_.Length/1KB) }:



* 1. Quebra de linha e arrays:
     1. Nosso código está muito grande para uma linha só, então decidimos quebrar ele em várias linhas para melhorar a visualização, além de usar os alias de cada comando para ficarem menores também:

gci -Recurse -File `

| ? Name -like "\*\_migrando\_\*" `

| Select `

Name, `

{ "{0:N2}KB" -f ($\_.Length/1KB) }

* + - 1. ?: alias do Where-object.
      2. Select: Alias do select-object.
    1. Colocamos o ‘**`**’ no final de cada linha, menos na última, para indicar para o PS que esse comando é um só e não é para executar uma linha por vez. Não colocamos na última porque ela é de fato a última linha.
    2. No final o resultado é o mesmo da última imagem.
    3. Entretando o backtick (acento crase ‘`’) é usado para as linhas que não temos o pipe (‘|’), pois, se colocarmos o pipe no final da linha o PS tem a inteligência de esperar o que vem na próxima linha para executar o primeiro comando, então, no final o código fica assim:

gci -Recurse -File |

? Name -like "\*\_migrando\_\*" |

Select `

Name, `

{ "{0:N2}KB" -f ($\_.Length/1KB) }

* + 1. Podemos colocar qualquer expressão ou whatever em variáveis então no nosso comando select podemos deixar assim:

$nameExpr = "Name"

$lengthExpr = { "{0:N2}KB" -f ($\_.Length/1KB) }

gci -Recurse -File |

? Name -like "\*\_migrando\_\*" |

Select `

$nameExpr, $lengthExpr

* + - 1. Declaramos o nome e a expressão em variáveis e substituímos no nosso código. E no final o resultado é o mesmo.
    1. A vírgula que colocamos para separar as variáveis no nosso select indica que estamos fazendo um Array com essas variáveis.
       1. Se colocarmos 1, 2, 3, 4 no powershell e verificarmos o tipo, veremos que é um Array:

(1,2,3,4).GetType().Name



* + 1. Se quisermos especificar que queremos um Array, colocamos um @ antes do ():

@().GetType().Name



@(1).GetType().Name

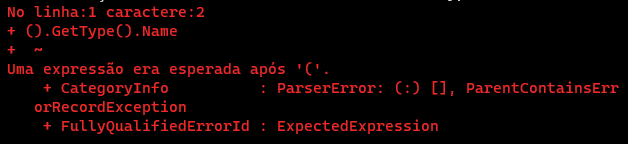


* + - 1. Sem o @:

(1).GetType().Name



().GetType().Name



* + - 1. Se for colocar 1 número só dentro do (), mas mesmo assim quiser forçar que é um Array, pode, além de colocar @, colocar uma ‘,’ antes:

(,1).GetType().Name



* + 1. Sabendo disso e que o select utiliza arrays, podemos criar outra variável e colocar ambas as expressões separadas por vírgula, formando um Array e deixando nosso código ainda mais limpo:
       1. Opções para dizer que é um Array:

$params = @($nameExpr, $lengthExpr)

$params = ($nameExpr, $lengthExpr)

$params = $nameExpr, $lengthExpr

* + 1. Agora nosso código completo ficou assim e o resultado final continua o mesmo:

$nameExpr = "Name"

$lengthExpr = { "{0:N2}KB" -f ($\_.Length/1KB) }

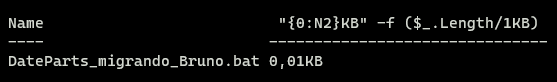
$params = $nameExpr, $lengthExpr

gci -Recurse -File |

? Name -like "\*\_migrando\_\*" |

Select `

$params



* 1. Usando HashTables no PowerShell:
     1. Nossa tabela está um pouco estranha, o nome está escrito em inglês e a coluna do tamanho do arquivo está muito estranha. Considerando isso vamos fazer as devidas alterações para visualizar melhor.
     2. Vimos que @() cria um array, mas se quisermos criar um HashTable utilizamos @{}:

@{}.GetType().Name



* + 1. Hashtables possuem diversos métodos e podemos verificar todos com o Get-member.
       1. Para adicionar coisas no hash utilizamos o .add(“Label”, “nome\_da\_label”) para colocar no título e para cada expressão o .add(“Expression”, “expressão\_da\_conversão\_do\_tamanho”). No final fica assim:

$nameExpr = @{}

$nameExpr.Add("Label", "Nome")

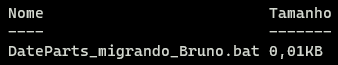
$nameExpr.Add("Expression", { $\_.Name })

$lengthExpr = @{}

$lengthExpr.Add("Label", "Tamanho")

$lengthExpr.Add("Expression", { "{0:N2}KB" -f ($\_.Length/1KB) })

* + - 1. Resultado da execução:



* + 1. Esse método para criação de hash não é boa pois utilizamos várias linhas e ficamos chamando o .add o tempo todo. Para melhorar nosso script ainda mais podemos fazer tudo isso na linha do @{}:

$nameExpr = @{

Label = "Nome";

Expression = { $\_.Name }

}

$lengthExpr = @{

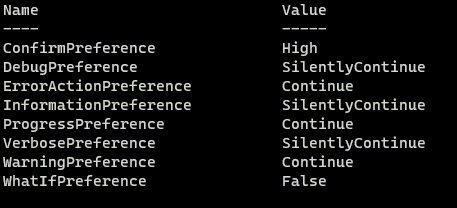
Label = "Tamanho";

Expression = { "{0:N2}KB" -f ($\_.Length/1KB) }

}

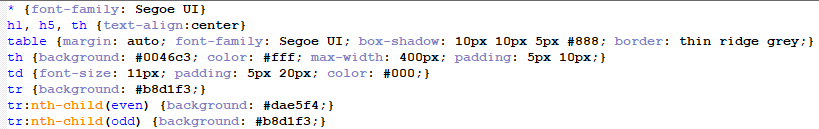
* + - 1. Se torna muito melhor e mais prático criar toda a hash de uma vez em uma única linha ao invés de ficar usando o add.

1. **Aula 5 – Nosso Script PS1:**
   1. Nosso primeiro script e variáveis de preferência:
      1. Para converter nossas anotações txt em script pra não ficar copiando e colando no terminal toda vez que quiser executar, basta salvar o arquivo como .PS1.
      2. Para comentar em script utilizamos #, como em python.
         1. <# e #>: Comenta várias linhas de uma vez, como ``` ``` em python.
      3. Existem várias variáveis de preferência durante a execução de um script e para encontrar elas podemos simplesmente usar o Get-Variabel | ? Name -Like \*preference\*:

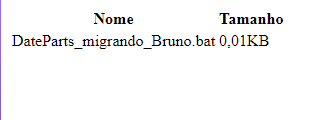


* + 1. Nosso PS vem por padrão com a preferência de ação de erro = continue, ou seja, se der algum erro no nosso script ele avisa no terminal, mas não para a execução do script, seguindo até o fim dele.
    2. Isso é muito ruim porque se algo der errado em uma parte e o resto precisar dessa parte para que funcione adequadamente, algo pode dar MUITO errado.
    3. Para alterar as variáveis de preferência podemos colocar no início do nosso script $ErrorActionPreference = "Stop", por exemplo, aí sempre que der algum erro no script ele irá parar a execução.
  1. Contextos de execução:
     1. O PS trabalha com contextos, ou seja, o contexto da janela do PS é diferente do contexto de um script, o script é um contexto filho da mãe OS.
     2. Considerando isso podemos imaginar que o contexto filho tem acesso a todas as variáveis do contexto mãe, entretanto, ele não pode alterar suas variáveis.
     3. Mas se quisermos que os contextos filhos não tenham acesso a uma variável do contexto mãe, colocamos a variável do contexto mãe como privada a partir do $private:variável.
     4. Ao tornar ela privada nenhum contexto filho terá acesso ao seu conteúdo.
  2. Gerando relatórios HTML:
     1. Para converter algo pra HTML utilizamos o ConvertTo-Html. Podemos passar vários parâmetros para ele como:
        1. -Head $styleTag:
           1. "<style> $estilos </style>"
           2. A variável estilos possuí o caminho do nosso arquivo css: $estilos = cat A:\Users\bruno\temp\styles.css.

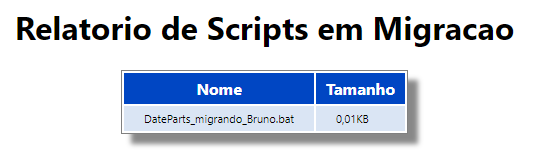
Conteúdo do css:



* + - 1. -Title $tituloPagina:
         1. "Relatorio de Scripts em Migracao"
      2. -Body $tituloBody.
         1. "<h1> $tituloPagina </h1>"
      3. Podemos colocar as variáveis dentro das tags e “” direto pois o PS tem a inteligência de saber que isso é uma interpolação, sem a necessidade de termos que colocar “<h1” + $tituloPagina + “</h1>".
    1. Essas variáveis criadas para o ConvertTo-Html foram para deixar a página web do relatório mais bonita e de melhor visualização, pois sem elas a página ficou exatamente igual ao PS:
       1. Sem as variáveis:



* + - 1. Com as variáveis:



* 1. Convertendo o relatório para JSON e CSV:
     1. Exatamente a mesma coisa que para o HTML:

$resultado | ConvertTo-JSON | Out-File A:\Users\bruno\temp\relatorio.json

$resultado | ConvertTo-CSV | Out-File A:\Users\bruno\temp\relatorio.csv

* + 1. Para fazer com que no precisemos criar 3 scripts de conversão alterando somente esse valor, podemos fazer uma verificação e pedir para que passem um parâmetro ao executar o script:

param($tipoDeExportacao) # Pede o parâmetro do tipo exportação. Deve ser a primeira linha do script

if ($tipoDeExportacao -eq "HTML") {

$estilos = cat A:\Users\bruno\temp\styles.css

$styleTag = "<style> $estilos </style>"

$tituloPagina = "Relatorio de Scripts em Migracao"

$tituloBody = "<h1> $tituloPagina </h1>"

$resultado | ConvertTo-Html -Head $styleTag -Title $tituloPagina -Body $tituloBody | Out-File A:\Users\bruno\temp\relatorio.html

} elseif ($tipoDeExportacao -eq "JSON") {

$resultado | ConvertTo-JSON | Out-File A:\Users\bruno\temp\relatorio.json

} elseif ($tipoDeExportacao -eq "CSV") {

$resultado | ConvertTo-CSV -NoTypeInformation | Out-File A:\Users\bruno\temp\relatorio.csv

}

* + 1. A conversão pra CSV faz com que o arquivo venha com a informação do tipo, por isso passamos o parâmetro -NoTypeInformation. Com ele nosso arquivo é convertido perfeitamente.