PowerShell Best Practices

Improving your automation skills!

Kamil Procyszyn - kamilpro.com

June 21, 2024

Contents

1 Intro **2**

2 Help **3**
2.1 Understanding syntax . . . 3
2.2 OK help is great and stuff, but what about A.I. - can I not just ask A.I. to do stuff for me? . . 3

3 Visual Studio Code **4**
3.1 But what about ISE? . . . 4

4 Transcript - A built-in logging function **5**

5 Read-Host - just don't use it **6**

6 Splatting **7**

7 Array vs List **8**

8 Error action preference **9**

9 Unary array expression or Return ,$array **11**

10 Errors handling of native applications **12**
10.1 Exit code . . . 12
10.2 PSNativeCommandUseErrorActionPreference . . . 12

11 A difference between an array and a hashtable **13**
11.1 Arrays . . . 13
11.2 Hashtable . . . 13
11.3 When to use array or hashtable . . . 13

12 Modules **14**
12.1 PSReadLine - Command history in the shell . . . 14
12.2 Pode and Pode.Web - Web server and web front-end . . . 14
12.3 Posh-Git - Git information in the prompt . . . 14
12.4 KpPwpush - Share your secrets securely . . . 15

1 Intro

Hello World… or Fellow Scripter should I say!

I'm glad to see you on this e-book of PowerShell best practices!

PowerShell often has many ways of achieving the same thing, and usually one is more performant than the other or simply creates a better experience.

In this very first version of the e-book, I have gathered the 10 items I teach to aspiring scripters most often, as well as seasoned developers. There are code samples provided, which you can just copy, paste and try yourself.

Hope you'll learn something new from the E-Book!

Kamil

2 Help

PowerShell has a great built-in help, right in the shell. The advantage of it is that you don't need to go out on the Internet to learn how to use a given function. First of all, run `Update-Help` to download all help content locally. Then you can use `Get-Help`, or its alias `help` to learn how to use a function. Or my favourite, `function name -?` Let's a have look at a few examples:

```
Get-Help Get-Process
help Get-Process
Get-Process -?
```

There are a few more options available:

```
help Get-Process -Full # Provides complete documentation
help Get-Process -Examples # Lists just examples
help Get-Process -Online # Open a webpage with documentation, if it's available.
```

2.1 Understanding syntax

Let's have a look now at the case of understanding help:

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

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

Get-Process -Id <int[]> [-Module] [-FileVersionInfo]
[<CommonParameters>]

Get-Process -Id <int[]> -IncludeUserName [<CommonParameters>]

Get-Process -InputObject <Process[]> [-Module] [-FileVersionInfo]
[<CommonParameters>]

Get-Process -InputObject <Process[]> -IncludeUserName
[<CommonParameters>]
```

There are a few things going on:

1. Each row starting with Get-Process is a different set name, there are so many mutually exclusive parameters, hence this row provides a different allowed set. E.g. `Name` parameter cannot be used together with `Id`parameter.

2. Parameters within [] are optional. If you look at the first set, Get-Process can be executed without any parameters.

3. The value after the parameter name indicates the object type. If there's no value after the parameter, it's a switch e.g. `-IncludeUserName`

2.2 OK help is great and stuff, but what about A.I. - can I not just ask A.I. to do stuff for me?

Yes, you can, that's why A.I. became popular. But especially when you're learning how to use the language, reading and understanding help will make you better at understanding PowerShell. Also A.I. in its current form might often provide wrong code snippets, and you might spend more time debugging it than actually writing code. Once you're familiar with how the shell works, A.I. works as a great work multiplier.

3 Visual Studio Code

Link: Visual Studio Code

PowerShell's official code editor of choice is VS Code (Visual Studio Code) and it has been for years. To have a full PowerShell experience, you need to install the PowerShell extension by going to the Extensions pane or opening any .ps1 file (VS Code will prompt you to install it when you open any file with .ps1 extension).

If you haven't heard about VS Code before, here's a quick pitch:

- Supports PowerShell 5 and 7
- Supports probably every programming language out there (as long as you can find relevant extensions)
- Cross-platform support (Linux, Windows, macOS)
- Git integration
- Thousands of extensions exist
- Dark and light themes out of the box
- Supports multiple terminal tabs
- GitHub codespaces uses VS Code in the browser
- Settings synchronisation
- Support for working in container/dev containers
- Baked in the tunnel for working remotely
- Dedicated debugging operations
- Great search capabilities
- And many more

3.1 But what about ISE?

Windows still comes with preinstalled Windows PowerShell ISE, which has been deprecated for years at this point - it only receives security patches. The main limitation is it only supports Windows PowerShell, that is version 5.1. Is it still worth using? I'd say, if you're on a machine which uses Windows PowerShell, and you're not allowed to install VS Code, then having ISE is better than having nothing. Otherwise, use VS Code, or any other modern code editor.

4 Transcript - A built-in logging function

PowerShell has a built-in simple logging function.

All needed to execute it is to run:

```
Start-Transcript
```

At this point, everything that gets displayed to the console is also saved to the transcript. By default, the transcript gets saved to Documents on Windows and the home folder on other systems. To stop the transcript:

```
Stop-Transcript
```

There are a few useful parameters:

- Path: Path to the file where to save the transcript
- OutputDirectory: Change the default directory
- Append: Adds transcript at the end of the file.
- Force: Tries to save to Read-Only file, changes permission to Read-Write

5 Read-Host - just don't use it

Read-Host is one of these cmdlets we tend to use when we need to get some information from the user. The problem with Read-Host is we can't validate what the user has provided, nor we cannot pass any information to it - so the script will always have this manual step.

There's a better way - using parameters.

```
param(
    [Parameter(Mandatory)]
    [int]$Number
)

Write-Host "$Number * $Number = $($Number * $Number)"
```

Ok so there are a few things going on, let's go in line by line: 1. param() - This indicates a parameter block, where parameters are going to be stored 2. [Parameter(Mandatory)] - This is a decorator for the argument, and it makes is Mandatory - that means PowerShell will prompt use to provide if not provided. In the script, the user could as well pass this like that: `Script.ps1 -Number 5` and the script would just run. 3. `[int]$Number` - This this is the actual parameter, of type integer - if any other type has been supplied, PowerShell will throw an error. Type is optional, it doesn't need to be provided.

And just like that, we have much more versatile code. Parameters can be used in the scripts as well as in functions. One requirement, the `param` block must be placed at the top of your script or function, otherwise PowerShell won't recognise it.

6 Splatting

Considering you have a cmdlet which takes a considerable amount of parameters, this example looks for ps1 files, in your home directory.

```
Get-ChildItem -Recurse -Path $HOME -Depth 2 -Name -File -Filter "*.ps1"
```

We can splat this code, which means to pass these parameters as a hashtable:

```
#Declare a hashtable with all required parameters
$params = @{
    Depth = 2
    File = $true #to pass switch, we set the value to $true
    Filter = "*.ps1"
    Name = $true
    Path = $HOME
    Recurse = $true
}
Get-ChildItem @params #to indicate it's a hashtable we use the @ symbol
```

Writing code with splatting makes it clearer to read. In version control, it makes it obvious which argument has changed, as the reviewer doesn't need to figure out which arguments have changed.

Since we have hashtable, we can now use all its benefits:

- We can add arguments, which come in handy with conditional statements. e.g. if a user has provided their email address, add it to arguments when creating a new user account
- We can change the values of current arguments
- We can remove keys from the hashtable

7 Array vs List

Before we get into any details, I want to run these two code snppets and compare results.

```
function Test-Array
{
    $array = @()
    foreach ($i in 1..123456)
    {
        $array += $i
    }
}
Measure-Command {Test-Array}
```

it took 115 seconds on my machine. Let's try it with List instead

```
function Test-List
{
    $list = [System.Collections.Generic.List[int]]::new()
    foreach ($i in 1..123456)
    {
        $list.Add($i)
    }
}

Measure-Command {Test-List}
```

0 seconds! And that's only some 100k of integeres, imagine if there was a complex object used, or we would go into millions of items in an array. So why the difference? Arrays are by definition not resizable, so we shouldn't be able to add any items to an array. Why does PowerShell then allow us to add items to an array with the += operator? It's because PowerShell is nice - what really happens behind the scenes, is: 1. PowerShell creates a new array 2. It copies the contents of the current array to the new array 3. Adds new items to the new array 4. Replace the old array with the new array That's a lot of work, and this takes more time the more items are than in an array.

This is why Generic lists exist - they are designed so that a lot of items can be added to them. We need to specify what can be added to the Generic list:

```
$data = [System.Collections.Generic.List[int]]::new() # int is type
$data.Add(5) # and this is how we add something to the list
```

Working with lists works the same as with the array.

8 Error action preference

PowerShell has a concept of non-terminating errors - that is when an error happens it will carry on running your script. Run this snippet:

```
& {
    Write-Host "Here 1"
    Write-Error "Ouch"
    Write-Host "Here 2"
}
```

All three lines run, despite there being an error. Let's change now the default behaviour to stop in case of an error:

```
& {
    $ErrorActionPreference = 'Stop'
    Write-Host "Here 3"
    Write-Error "Ouch 2"
    Write-Host "Here 4"
}
```

The script stops on the error, great! But what if we want to handle the error - that is do something in case it happens so that the script can carry on? You might have heard about Try/Catch, let's try it with default settings:

```
& {
    try
 {
        Write-Host "Try 1"
        Write-Error "Try 2"
        Write-Host "Try 3"
 }
    catch
 {
        Write-Host "Catch 1"
 }
}
```

With default settings, the catch isn't executed at all, hence error is not handled. Let's now tell PowerShell to stop in case of an error:

```
& {
    $ErrorActionPreference = "Stop"
    try
 {
        Write-Host "Try 4"
        Write-Error "Try 5"
        Write-Host "Try 6"
 }
    catch
 {
        Write-Host "Catch 2"
 }
}
```

Now the moment the error happened, PowerShell stopped executing the Try block and moved to the Catch block.

If you're wondering what: `& {}` is, is called a script block - and the reason to use it is to not change your

settings. Script block runs in its own scope, and e.g. change to a variable like $ErrorActionPreference within script block doesn't change the variable setting globally.

9 Unary array expression or Return ,$array

PowerShell is helpful, and sometimes it is helpful too much. One such example is when we want to return an array of items. When there is more than 1 item in the array - we get an array. But when there's only a single item in the array, PowerShell returns that single item instead of the array - which might cause issues further down the line when we explicitly want to work with an array. Consider this example:

```
function Get-ArrayWithItems
{
    $output = @(
        1,2,3
 )
    return $output
}

$data = Get-ArrayWithItems #this is an array
Write-Host "Multiple item type: $($data.GetType().BaseType)"

function Get-ArrayWithSingleItem
{
    $output = @(
        1
 )
    return $output
}
$data = Get-ArrayWithSingleItem #this is an integer!
Write-Host "Single item array type: $($data.GetType().BaseType)"
```

To always get an array, simply add a comma after the return statement:

```
function Get-ArrayUnrary
{
    $output = @(
        1
 )
    return ,$output
}
$data = Get-ArrayUnrary #this is an array
Write-Host "Unrary return type: $($data.GetType().BaseType)"
```

10 Errors handling of native applications

Native applications usually return the exit code - not the actual error itself. There are two main techniques we can employ to handle their execution

10.1 Exit code

By the rule of thumb, successful execution exits with 0, and any non-0 exit code is considered a failure - you'd need to check with application documentation to validate what each exit code means.

PowerShell has a special variable `$LASTEXITCODE` which holds the value of the last exit code.

The examples below assume you have Git installed.

```
git --version
if ($LASTEXITCODE -ne 0)
{
    throw "Unable to check Git version"
}

git bla
if ($LASTEXITCODE -ne 0)
{
    throw "Something wrong"
}
```

10.2 PSNativeCommandUseErrorActionPreference

Introduced in PowerShell 7.4, we can change how PowerShell treats application errors and allows us to handle them like a normal function

```
& {
    $PSNativeCommandUseErrorActionPreference = $true
    try
    {
        git --version
        git bla
    }
    catch
    {
        throw
    }
}
```

11 A difference between an array and a hashtable

On the surface, the declaration of array and hashtable are quite similar:

```
$array = @()
$hashtable = @{}
```

but those are entirely different data structures.

11.1 Arrays

- Arrays are quick at iterating over them, hence any loop going from the first item to the last will go quickly.
- Arrays allow retrieving specific items by their index e.g. `$array[3]`
- You can retrieve the first item in the array with`$array[0]`
- You can retrieve the last item in the array with `$array[-1]`
- Arrays allow for duplicate items in them `@(3,3,3,3,3)`
- Arrays by definition are not meant to be resized (please read a chapter about arrays and lists) and the way PowerShell adds items to the array is costly - the performance hit starts to be noticeable in areas of a few thousand items.
- PowerShell allows to mix of any sort of items in the array e.g. `@(78, "Hi", $true)` is a valid array
- In general, arrays are quick at retrieving data.

11.2 Hashtable

- Hash stores items in key-value pairs e.g. `@{3 = "three"}`
- Keys must be unique in the hashtable
- Adding items to hashtable is fast
- Keys can be added or modified by providing their name

```
$hash = @{}
$hash["PowerShell"] = "Is best" #creates a new key called PowerShell
$hash["PowerShell"] = "Is The Best" #updated key PowerShell"
```

- Values can be retrieved by their name
- To loop over hashtable, a dedicated `.GetEnumerator()` method is available

11.3 When to use array or hashtable

- If values must be unique, use a hashtable. There's `.ContainsKey()` method to check whether a key with a given name already exists
- If values must be retrieved by a specific name, use a hashtable
- If all required is to loop over data, then an array might be a good choice. Look at a list if adding values to an array is required

12 Modules

PowerShell has a great built-in function, but often it's just not enough. When we need extra functionality, like building a website or improving our shell experience - we can extend PowerShell by this. Modules here are the ones I've used or still use on a daily basis - hope you find something useful!

12.1 PSReadLine - Command history in the shell

Documentation

PowerShell Gallery

This is one of the modules which you just need to install and it works in the background (but you can get so much more out of it if you're willing to spend some time on configuration!)

Simply run `ps1 Install-Module -Name PSReadLine`, restart your shell and… enjoy autocompletion:

Figure 1: PSReadLine providing automation

12.2 Pode and Pode.Web - Web server and web front-end

Documentation - Pode

PowerShell Gallery - Pode

Documentation - Pode.Web

PowerShell Gallery - Pode.Web

Pode is a PowerShell web server - this is great for writing a backend, or if for some reason you can't use Azure Functions. My love with Pode happened in its second module - Pode.Web, this is an extension to Pode, which allows you to write Frontend - or Website - entirely in PowerShell. If you're familiar with PowerShell scripting, having an application written in Pode.Web takes a couple of hours. I have a couple of videos which hopefully you'll find useful:

Video 1

Video 2

12.3 Posh-Git - Git information in the prompt

Documentation

PowerShell Gallery

This one is invaluable if you use Git and like working in a terminal. Posh-Git simply puts information about the current Git Status right in the prompt. To have a basic setup going:

```
Install-Module -Name Posh-Git
Add-PoshGitToProfile
```

Restart your terminal, and the next time you open a Git repository in the terminal, you'll see an updated prompt:

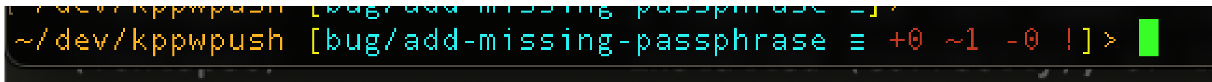

Figure 2: Posh-Git Prompt

12.4 KpPwpush - Share your secrets securely

Documentation

PowerShell Gallery

Sending secrets to others is... challenging. Or uncomfortable. What if you could send a link to the other person, and that link expires after they've seen your secret? This is what pwpush.com does. Even if the link gets to some dishonest people later on, it's worth noting, as the secret has already been purged.