MSAdministrator

All about Windows Automation

Powershell Console, Scripts, Functions, Modules, Cmdlets, Oh My!

AUGUST 19, 2018 / POWERSHELL / JOSH RICKARD

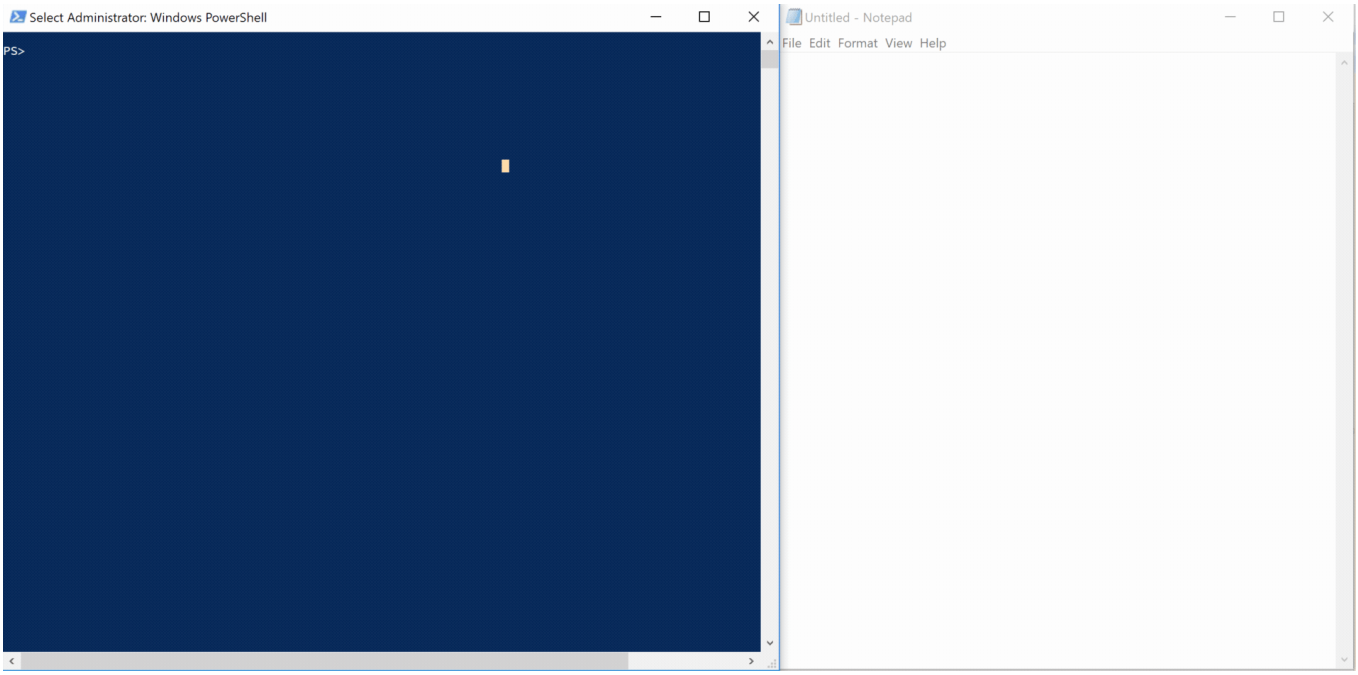
PowerShell is unique, but it is also necessary in today's world. I find that newcomers to the language are sometimes confused or don't understand the basic layers of PowerShell scripting. You can use PowerShell in different ways, but I have yet to find an all encompassing article that explains the high-level of how to interact with PowerShell in these different ways.

This post will explain the different concepts (or layers) of PowerShell code organization and their main advantages and use cases. We will dive into the different approaches of implementing PowerShell and why you would use one over the other.

Console

The PowerShell console is a Windows shell environment that is used to invoke the PowerShell engine. The console is traditionally used when you want to run one-off commands or to invoke other custom scripts, functions, modules, cmdlets, etc. A typical PowerShell `developer` will use the console to quickly do a single action. For example, if you wanted to identify running processes on a computer you would run the `Get-Process` CmdLet. You probably would not write a script or function to invoke this simple command unless you wanted to do further action against a process or format the output.

The PowerShell console is great for these quick and dirty commands, but a PowerShell developer often times does not want to write tons of logic in the console. Instead, you would probably write that logic in a script or function. I consider the use of the PowerShell console as a quick way of proving that your thoughts or logic will work and not necessarily the final product. If all you need to do is `Get-Process -Name notepad | Stop-Process` then just typing that in the console is all you need. Great! If you need a little more logic or flow control in your efforts then you will probably choose to write a PowerShell script or function.



Scripts

PowerShell scripts are placed into a `.ps1` file type. These scripts usually contain a logic flow of multiple lines of code that the caller needs to perform. These may contain loops or if statements; but they do not have to. Think of the steps that you need to perform to build a cake or a car engine. Each of these steps are usually outlined, and the steps are a recipe. In PowerShell these are found in a `.ps1` file.

With PowerShell scripts, you will invoke them directly in the PowerShell console. Invoke is really a fancy way of saying you will call the file directly. For example, if you have a script that traverses files within a folder and does an action when a file or folder matches the name you're looking for. You may put this in a script, since it may be something you do frequently or want to save it for someone else to run.

```
$filesAndFolders = Get-ChildItem -Path ~/Desktop/SomeFolder -Recurse

foreach ($fileOrFolder in $filesAndFolders) {
    if ($fileOrFolder -contains 'some name') {
        Write-Output "Found $fileorFolder"
    }
}
```

Let's say that this code above was in a file named `getFileAndFolders.ps1`. When you open up the Powershell console you would typically navigate to the location this script is housed and invoke the script directly.

```
.\getFileAndFolders.ps1
```

This is the typical way that PowerShell scripts are used. They are basic, but may have some logic. They are designed to be invoked and they are easy to use. It's much easier to tell someone to open up PowerShell and run the file then it is with more advanced methods like Functions and Modules.

Basic Functions

In PowerShell there are two different levels of functions; a basic function and a Advanced Function. I will explain both in detail, but both are "wrappers" for specific code that you want to use. It is best practice to **have a function do 1 thing and do that 1 thing well!** Functions are used in many different cases but they are created so that you can re-use the code in that "wrapper" throughout your script or module.

A "basic" function is just that - basic. You can tell if code you are looking at is a function by the declarative statement of `function`. A basic function is one that you can specify 0 or more parameters (or values) that the user passes into the function. For example, below are two separate "basic" functions. The first takes no parameters and is the same code above but wrapped in the `function` statement.

```
function getFileAndFolders () {
    $filesAndFolders = Get-ChildItem -Path ~/Desktop/SomeFolder -Recurse

    foreach ($fileOrFolder in $filesAndFolders) {
```

```
        if ($fileOrFolder -contains 'some name') {
            Write-Output "Found $fileorFolder"
        }
    }
}
```

That's it! This is a basic function; easy huh! Where things get a little challenging for newcomers it that if you call this file like you did previously, the output will be different (actually no output).

```
.\getFileAndFolders.ps1
```

But why? When your `.ps1` is invoked you are calling the same file but with `functions` you are not actually invoking the code, but rather you are loading it into memory. This means that you now must call your function directly. To do this, in a PowerShell console type `getFile` and the `Tab` key. You will see that PowerShell knows what your function name is and now you can call it directly without always having to call the script file.

But, **a HUGE but**, this is only for this session of PowerShell. As soon as you close your current PowerShell console then it will no longer be in memory and you will have to re-import/invoke your script file again to load it into memory.

Secondarily, you can also load this function into memory by dot sourcing it. What this means is that you are telling the console that you want to load the file into memory but not actually invoke it or run the code. You do this by calling the following:

```
. .\getFileAndFolders.ps1
```

It doesn't seem like that big of a difference, and it is almost identical except for the single `.` and space before the path string to the file you want to load into memory. Dot sourcing your code means you want to use it coming up, but you do not necessarily want to invoke the code at this time.

There are big advantages to this, but we will explain that a bit later.

To continue with this first example, you can, within your `.ps1` script invoke the function directly after loading it into memory. For example, in your `getFileAndFolders.ps1` you can call your function right after you have loaded into memory.

```
function getFileAndFolders () {
    $filesAndFolders = Get-ChildItem -Path ~/Desktop/SomeFolder -Recurse

    foreach ($fileOrFolder in $filesAndFolders) {
        if ($fileOrFolder -contains 'some name') {
            Write-Output "Found $fileorFolder"
        }
    }
}

# Calling the function above that has been loaded into memory
# You must load your function into memory first before calling it, so.
# The order of your code MATTERS!
getFileAndFolders
```

This way of invoking your script means that you still take advantage of a function but you are still invoking the code within the same file. This helps when you want to segment your code into blocks (functions) but still have a single file that a user may invoke at any point. Please note, that you must have your `function` statement before you actually invoke it. This is from top to bottom within your `.ps1`.

Let's use this same example, but let's add a simple parameter. Let's change our function to pass in the folder you want to search within. When using a basic PowerShell function, you would add a parameter by adding a variable in between the parenthesis.

```
function getFileAndFolders ($SearchFolder) {
    $filesAndFolders = Get-ChildItem -Path $SearchFolder -Recurse

    foreach ($fileOrFolder in $filesAndFolders) {
        if ($fileOrFolder -contains 'some name') {
            Write-Output "Found $fileorFolder"
        }
    }
}

# Calling the function above that has been loaded into memory
# You must load your function into memory first before calling it, so.
# The order of your code MATTERS!
getFileAndFolders ~/Desktop/SomeFolder
```

As you can see, we add the `$SearchFolder` parameter to our function, but we also replaced the `/Desktop/SomeFolder` string in our function with that variable. We did this, because a parameter's value that is passed into a function will replace that value in your function. So, when we call our function `getFileAndFolders` with a value of `~/Desktop/SomeFolder`, then our function will take that value and replace our variable of `$SearchFolder` with that value through out your function.

Advanced Functions

Basic PowerShell functions are great for basic operations. That may be all you need, but as you get more familiar with PowerShell and want to create extremely durable and re-usable code that others will use, then you will want something more advanced. There's a secret to advanced functions, but it's not difficult and it adds a lot of great functionality for free!

```
function getFileAndFolders {
    [CmdletBinding()]
    Param(
        $SearchFolder
    )

    $filesAndFolders = Get-ChildItem -Path $SearchFolder -Recurse

    foreach ($fileOrFolder in $filesAndFolders) {
        if ($fileOrFolder -contains 'some name') {
            Write-Output "Found $fileorFolder"
        }
    }
}

# Calling the function above that has been loaded into memory
# You must load your function into memory first before calling it, so.
# The order of your code MATTERS!
getFileAndFolders ~/Desktop/SomeFolder
```

That's it! Did you see what I did there? There's a special indicator called `[CmdletBinding()]` that enables a whole slew of features to your function. You now have an advanced PowerShell function!! Look at you go!

The great thing about `CmdletBinding` is that you now have a lot more flexibility and you can help users of your function by ensuring that they are operating within your defined requirements.

So, I somewhat lied above; you did create an Advanced Function but using it this way is not going to help you much. For example, if you call our `getFileAndFolders` function with NO input value then you will get a bunch of red (errors). That sucks, but with `[CmdletBinding()]` we have an arsenal of ways to validate and require certain values to meet our demands. If we want to require that all users pass in a value for `$SearchFolder` then we can force that. We can even ensure that the passed in value is a `[string]` and not an integer or some other value type that we are not prepared to handle.

```
function getFileAndFolders {
    [CmdletBinding()]
    Param(
        [Parameter(Mandatory=$true)]
        [string]$SearchFolder
    )

    $filesAndFolders = Get-ChildItem -Path $SearchFolder -Recurse

    foreach ($fileOrFolder in $filesAndFolders) {
        if ($fileOrFolder -contains 'some name') {
            Write-Output "Found $fileorFolder"
        }
    }
}
```

We added some definitions around our parameter to ensure that the parameter must `(Mandatory)` be passed used when calling this function. We did this by setting `Mandatory` to `$True`. We also `casted` our variable as a `[string]` to ensure that any value that is passed into that parameter must be a string. These two definitions of our parameter ensure that we handle all situations that meet our requirements.

There are lots of different validations that can be set on a parameter. I'm not going to go into these at this time, but I will share an example of what an advanced parameter validation may look like:

```
[Parameter(Mandatory = $true,
           ValueFromPipelineByPropertyName = $true,
           Position = 0)]
[ValidateNotNull()]
[ValidateNotNullOrEmpty()]
[ValidateScript( {
        If (Test-Path $_) {$true}else {Throw "Invalid path given: $_"}})]
[string]$SearchFolder
```

Along with having the ability to set expectations around what values can be passed into your function, you get a bunch of additional features with `CmdletBinding`. If you have ever used any other PowerShell built-in CmdLet's then you have probably noticed that there are additional "common" parameters that you have access to. With `CmdletBinding` you now have access to these as well within your function. Here is a list of these parameters, but I won't explain these in this post:

- Debug
- ErrorAction
- ErrorVariable
- InformationAction
- InformationVariable
- OutVariable
- OutBuffer
- PipelineVariable
- Verbose
- WarningAction
- WarningVariable
- WhatIf
- Confirm

With either a basic or advanced function, you must load them into memory before calling them. Both have their place in PowerShell scripting, but it is definitely best practice to use Advanced Functions when you can. They reduce potential errors when people are using your code and provide great documentation right out of the box.

Modules

There is some debate on what is considered a PowerShell Module, but I will explain it from my experience. A PowerShell Module is one that contains a `.psm1` (PowerShell Script Module) and a `.psd1` (PowerShell Module Manifest). They DO NOT need to contain both, but at minimum they contain a `.psm1`. A Script Module allows the user of the module to import it using the `Import-Module` CmdLet. If you place your `.psm1` in a PowerShell script path then PowerShell can also automatically load your module when you want to call a specific function. This means you don't have to worry about loading it into memory, the PowerShell engine takes care of this for you. This is a huge benefit and the users of your Module don't have to work magic to use your code.

PowerShell Modules can contain 1 or more functions. If you have a library of functions that you use for a single topic then you would group them into a PowerShell Module. I have a few on my *GitHub* and they are somewhat (I hope mostly) organized into a specific functionality or "product". For example, I have a PowerShell Script Module to interact with the ZenDesk ticketing system API. There are 44 PowerShell functions within this Module but they are all contained within a single PowerShell Script Module since they all relate to the ZenDesk ticketing system API.

Instead of having to load each function into memory individually, I can simply download the PowerShell Script Module and run `Import-Module -Name PoshZD` and it will load all of them into memory. This saves me and user of my Module a lot of time, plus it is easy to do!

Here is an example of how use a `.psm1` file to import all my functions:

```
# using module .\TemplatePowerShellModule\Class\TemplatePowerShellModule.Class1.psm1
# Above needs to remain the first line to import Classes
# remove the comment when using classes

#requires -Version 2
#Get public and private function definition files.
$Public = @( Get-ChildItem -Path $PSScriptRoot\Public\*.ps1 -Recurse -ErrorAction S
$Private = @( Get-ChildItem -Path $PSScriptRoot\Private\*.ps1 -Recurse -ErrorAction

#Dot source the files
Foreach ($import in @($Public + $Private)) {
    Try {
        . $import.fullname
```

```
    }
    Catch {
        Write-Error -Message "Failed to import function $($import.fullname): $_"
    }
}

Export-ModuleMember -Function $Public.Basename
```

With this in my `.psm1` file, any functions I have separated out into sub-folders named `Public` and `Private` will be imported automatically when you import my `.psm1` Script Module file.

Please note, I do have a Template PowerShell Module that you can use as reference if you would like to see how I organize my Script Modules

The big thing to take away from this section is that PowerShell Script Modules typically do not have compiled code or they are NOT written in C#/.NET. Yes, you can use `.psm1` and `.psd1` to import those compiled DLLs or code, but these are (IMO) CmdLets.

CmdLets

PowerShell CmdLets are compiled code. There are several nuances though that you should be aware of. The PowerShell engine provides built-in CmdLets as part of the PowerShell language framework. These are the source CmdLets and they are written in languages like C#/.NET, C++, etc. There are also community driven (and Microsoft written) CmdLet's that are compiled and can be written in the same languages, but they are not automatically installed on your computer as part of the PowerShell engine. Both can be called CmdLets, but there is a difference. The PowerShell engine CmdLet's do not require you to download them from a secondary location. These extra libraries or CmdLets require that you install them through a package manager or from a website.

The biggest benefit of CmdLets or compiled code in general is that they have access to lower level languages like C, C++, .NET, etc. and do not necessarily need the entire CLR (Common Language Runtime) to run their code. They are MUCH improved in speed since they do not have a layer of abstraction in front of them and can typically access the libraries (even hardware) directly.

You can identify PowerShell CmdLets because they have `.dll` files instead of `.ps1` functions. DLLs are compiled and have layers of classes and methods that allow you to reference them, but

many of the internal methods are `private` and do not allow you to access them directly.

Wrap-Up

That's it! PowerShell can be implemented in many different ways, but it ultimately depends on your need at the time of writing your code. I have forced myself to always write Advanced Functions because I never know if I am going to share the code I write with someone else or need to use it within another function or piece of code. It allows me to manage my code in a more organized manner, and it's definitely best practice among PowerShell enthusiasts.

Just remember that PowerShell Scripts are perfectly fine if that is what you need, but if you plan to ever share the code with someone else then put it into a Function. If you have multiple functions that are similar then put them into a Module. Lastly, if you need access to lower level APIs and libraries or have a need for speed, then PowerShell CmdLets are the way to go for sure.

scripts / functions / modules

LATEST POSTS

How to Setup a Hugo Website on GitHub

Using Amazon Sqs With Powershell

CATEGORIES

SOCIAL MEDIA