An Introduction to PowerShell for a DBA

[Alex Chamchourine](https://www.sqlservercentral.com/author/Alex-Chamchourine), 2020-05-01 (first published: 2017-12-21)

FROM: <https://www.sqlservercentral.com/articles/an-introduction-to-powershell-for-a-dba>

All of us who worked with multiple programming languages over the span of our careers, will probably agree that the most frustrating thing about learning any new language are the first steps. The simplest things such as “should I add a terminator after that line?” or “do I really need this odd looking symbol in front of that variable?” keep getting in the way. After you’ve gotten the hang of that, the second wave of frustration comes caused by the fact that’s not all the things are as simple as they look on the surface. Like when you do everything according to the manual and the result is not what you expected, for instance, due to a subtle environment setting.

Some time ago I was tasked with a writing a number of PowerShell (PoSh) scripts covering multiple aspects of SQL Server management. It was an interesting experience and despite all the ups and downs I have had fun. But given my share of frustration and Internet searching, I thought: what if I compile a short introduction that will save you some time and effort?

So here we go. To follow this guide, you will need the Windows PowerShell ISE or any other similar tool of your choice. The ISE comes standard with Windows. Just paste the snippets into the tool window and run them. Let’s start with a very basic stuff. First, of course, we want to know how to declare and use variables. [In PoSh, this is simple](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_variables): anything starting with a symbol “$” is a variable. For instance:

# Our first line of code! (You probably have figured out that this is a comment line, if you need multi line comments use <# ... #>)

$v = 1

PoSh will try to infer the type from the expression but you can give it a hint by using a type:

[string] $s = "1"

You have a few ways to express a string:

$s0 = "this is a string"

$s1 = '$s1 is a string'

$s2 = "$s1 too"

$s3 = @"

And this: $(Get-Date -Format "yyyy-MM-dd")

is a string

as well!

"@

Write-Host "`$s0=$s0"

Write-Host $s1

Write-Host $s2

Write-Host $s3

Write-Host " first line `n`r second line"

You see the difference between a single-quoted and a double-quoted string: PoSh will try to parse the double-quoted string by replacing the expressions starting with $ inside it with their values, while a single-quoted string will be used “as-is”. If you need to use “$” inside a double quoted string as a character escape it with the “grave accent”. Note that the “grave accent” has some other useful applications as well. Want to split your string into lines? Use “`n`r”! To express a multi-line string use the symbol “@”. Note that the symbol only can be followed or preceded by the “end of line” characters.

You probably noticed “Write-Host” and “Get-Date”. It’s what in PoSh called a cmdlet. You can think of it as of a pre-built function and PoSh has plenty of them.

Need to quickly check whether a string is an empty one (or a collection is empty or an object is null)? Simply check if it’s true or false:

[string] $empty

[string] $notempty = "qwert"

if(!$empty)

{

Write-Host '$empty is' "Empty"

}

if($notempty)

{

Write-Host '$notempty is' "Not Empty"

}

If you want to declare an array (which is a very versatile data type and you probably will want to use it a lot) there are a few ways. If you have all the elements beforehand:

$a = @("x", 1, 2)

Write-Host "First element = $($a[0]); Second element = $($a[1]); Third element = $($a[2])"

The combination @() defines an array. Note that a PoSh array is a zero based one. The type of an array again will be inferred, or you can provide it in the declaration. If you don’t know what will be in your array (i.e. a dynamic one) you can declare one and populate it later:

$a = @()

$a += "a"

$a += "B"

$a += "c"

Write-Host $a

Keep in mind that this approach, though clear and simple, could also be very expensive, as your array has to be destroyed and re-created every time you add a new element to it. It is better if you define the size of your array in advance. For instance, the next expression will declare an array of 3 elements with “a” as the default value:

$a = @("a")\*3

Write-Host $a.Count

Write-Host $a

This is a very short but not exactly clear syntax. My choice would be to use a New-Object cmdlet:

$a = New-Object string[] 3

Write-Host $a.Count

Let's declare a multi-dimensional one. First by using one array declaration inside another one (compare the following two declarations):

$a1 = @("x", 1, 2)

Write-Host "First element = $($a1[0]), type = $($a1[0].GetType()); Second element = $($a1[1]), type = $($a1[1].GetType())"

$a2 = @("x", @(1, 2))

Write-Host "First element = $($a2[0]), type = $($a2[0].GetType()); Second element = $($a2[1]), type = $($a2[1].GetType())"

And now by using a bit different syntax:

$a = New-Object string[][] 3,2

Write-Host $a.Count $a[0].Count

Note “New-Object” - a very useful cmdlet that creates an instance of any provided type (i.e. some object). PoSh has a shortcut to an object creating but not any object. It is a hash table. Don’t be alarmed by the name, if you need an object to use as a container for a group of features (a structure essentially) it is the way to go. You simply declare it:

$h = @{}

Write-Host $h.GetType()

and then add your features to it as you go:

$h.x = 1

$h.y = 2

Write-Host $h.x $h.y

What could be simpler? Of course, you can use it in a more formal way. It’s a collection of key-value pairs after all.

$h = @{}

$h.Add("X", 1)

$h.Add("Y", 2)

Write-Host $h.x $h.y

In addition to user-defined variables PoSh has quite a few pre-defined, or “auto” variables. The most commonly used are probably $true, $false and $\_ (yeah, just like this). The first two are obviously the boolean constants, true and false. The last one is the current pipeline object variable. Whenever you are dealing with a pipeline this variable is the way to access the current pipeline object:

$root = ${env:ProgramFiles(x86)} +"\Microsoft SQL Server\"

$files = (Get-ChildItem $root -Recurse) | Where-Object {$\_.FullName -like "\*sqlpackage\*"}

foreach($file in $files)

{

Write-Host $file.GetType() $file.FullName

}

Let’s have a closer look at this code snippet. There are a few new and important things here. First you see a system (environment related in this case) variable inside the curly brackets. The curly brackets are used when your variable name does not exactly follow the naming rules (that is it has some forbidden characters, like spaces for instance). Next is another PoSh cmdlet Get-ChildItem. This works on any container (like a file system directory or a certificate store), and is an extremely useful cmdlet. “–Recurse” is a parameter of type “switch” (we will talk about it more later).

After the cmdlet you see a pipeline character and that is indeed a pipeline – easily my favourite feature of PoSh! As most of the PoSh cmdlets support a default pipeline parameter, with this feature you can effortlessly connect multiple cmdlets building a complex workflow. Note how the Where-Object cmdlet receives the results of a directory scan and will apply your custom filter expression (the one in curly brackets) to them. In this case the curly brackets define an anonymous function which returns true or false.

Here you see one of our pipeline variable properties is being compared with some expression. PoSh has [a few other comparison operators](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_comparison_operators): “-eq” (=), “-ne” (!=), “-gt” (>), “-lt” (<), “-ge” (>=), “-le” (<=). Yes, they are a bit different from ones used in other languages but it’s easy to get used to. Note that “-like” can use only a wild character. If you want to unleash a power of RegEx use “-match”.

Also, to make our lives easier all of them are case-insensitive. To make them case-sensitive simply add a prefix “c” to them (that is for instance “-qe” becomes “-ceq”). And now, the last one in the snippet: “foreach” loop statement. Which is easily one of the most used features in PoSh as almost everything returned or received by PoSh cmdlets is a collection!

All script languages are notorious in being difficult to structure your code in case you need more than a few lines of it. The PoSh is not an exception but it has a few features to help.In PoSh you can define your own functions, pass parameters into and return results from, thus isolating the specific logic from the rest of the script. Have a look:

Function MyFunc

{

Param(

[Parameter(Mandatory=$true)] [string] $p1,

[Parameter(Mandatory=$false)] [string] $p2 = "This will be by default",

[Parameter(Mandatory=$true)] [ValidateSet("A", "B", "C")] [string] $p3,

[Parameter(Mandatory=$true)] [AllowNull()] [AllowEmptyString()] [string] $p4,

[Parameter(Mandatory=$true)] [AllowNull()] [AllowEmptyCollection()] [string[]] $p5,

[Parameter(Mandatory=$false)] [switch] $On

)

Write-Host "p1=$p1" "p2=$p2" "p3=$p3" "p4=$p4" "p5=$p5" "On=$On"

}

$a = @()

MyFunc -p1 "1" -p3 B -p4 $s -p5 $a –On

Though it looks rather wordy, I do strongly recommend you to use this thorough way of defining and calling functions. They will be much easier to understand some time after you created them. The list of parameter types shown in the snippet is a good one to start with. Everything in square brackets (except data type) is a parameter attribute. “Mandatory” is self-explanatory, “ValidateSet” means that the value must belong to the pre-defined set. “AllowNull”, “AllowEmptyString” and “AllowEmptyCollection” change the parameter validation behaviour.

Try to run the snippet without them to see the difference they make. As they might not look that important in the case of a simple script, imagine having an utility function that could be called deeply nested inside others. In this case you have no guarantee that your parameters won’t be empty strings or collections. Even if you have every bit of code to deal with them inside your function, PoSh will throw an error before it gets there. Finally, the last parameter is of a special type “switch”. It behaves like a boolean one but does not require a value. The very presence of such in a function call is enough – which makes your code more concise. You can find more on [parameters here](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_functions_advanced_parameters).

Remember I mentioned the pipeline as my most favorite feature? So, how to write a function to work with it? Easily:

function foo

{

Begin

{

Write-Host "Begin the processing"

}

Process

{

foreach($x in $\_)

{

Write-Host "Pipeline object" $x

}

}

End

{

Write-Host "End of processing"

}

}

1,2,3,4,5 | foo

foo

In this snippet we are first sending an array to the function via the pipeline and next calling the function without the use of the pipeline. You are already familiar with the variable “$\_”. Note the “begin”, “process” and “end” blocks. Those are so called input processing methods. The “begin” and “end” gets called only once in the beginning and the end of the function execution respectively and the “process” gets called for each object in the pipeline. If the function does not have any of them it’s the “end” block by default. You can find more on it [here (search for Piping Objects to Functions)](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions?view=powershell-5.1).

If you need to return something from a function, you naturally will use the “return” statement. There is an oddity about the way it’s done in PoSh though. First, it will return any output generated inside the function in addition to the return statement. As innocent as it sounds it could give you a real grief:

Function Foo

{

[boolean] $s

$s = "x"

$i = 1

return $i

}

$x = Foo

Write-Host $x.GetType() $x

Think that the function Foo will return 1? Well, think again, or better, run the snippet. It’s an array with two elements “False” and “1”! The declaration statement outputs the default value assigned to the declared variable and PoSh combines it with the value returned by the return statement. Thus the result is the array.

To my knowledge the only way to prevent such behaviour is to use “| Out-Null” after each such statement. It’s not pleasant but seems doing the trick well. The other oddity is the way it handles empty collections or collections with one element only. It unpacks the collection and returns its only element (instead of the whole collection) or returns null if the collection is empty. Run the following snippet:

function Return-Array1

{

$a = @("xyz", "b", "c")

Write-Host "Inside Return-Array1:" $a.GetType()

return $a

}

function Return-Array2

{

$a = @("xyz")

Write-Host "Inside Return-Array2:" $a.GetType()

return $a

}

$r1 = Return-Array1

$r2 = Return-Array2

Write-Host "`$r1 is still an array ($($r1.GetType())) of $($r1.Count) elements with $($r1[0]) as first one"

Write-Host "`$r2 is not an array anymore ($($r2.GetType())) and the first element is $($r2[0])!"

See how the type of the returned object has changed from array to string? To sweeten the pill, PoSh allows you to use “count” and “foreach” loop on pretty much any object (regardless whether it’s a real collection or not). Whenever you loop through the results, it will work as expected.

OK, enough about the return woes, back to code-organizing practices. Another trait of the well-organized code is a proper use of scopes. PoSh will manage it for you. Run the following snippet:

function Test1

{

$v1 = 1

$v2 = 2

$local = Get-Variable -Name v1 -Scope local -ValueOnly

$global = Get-Variable -Name v1 -Scope global -ValueOnly

Write-Host "Inside Test1: v1(local)=$local v1(global)=$global"

Test2

return

}

function Test2

{

Write-Host "Inside Test2: v1=$v1 v2=$v2"

return

}

$v1 = 4

Write-Host "v1 before the call:" $v1

Test1

Write-Host "v1 after the call:" $v1

By default, a variable created in a scope is visible within it and any child scope. However, if you try to change a variable within a child scope PoSh will hide the original one providing you with a local one of the same name. You can find more on scopes [here](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_scopes).

As one more way of managing your code PoSh supports [Try/Catch/Finally](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_try_catch_finally) blocks. You can and should use it (but not without some fine tuning). Please run the following snippet:

#$ErrorActionPreference = "Stop"

try

{

Get-ChildItem "x:\xxx" -Recurse

}

catch [System.Management.Automation.DriveNotFoundException]

{

Write-Host $\_.Exception.Message

}

Error! PoSh did not intercept it despite the try/catch block. This is because of the block catches only “terminating” (severe enough to stop the script execution) errors and the one happening here is not considered severe enough. Like I said, some fine tuning is required. Uncomment the top statement and run the snippet again. This time the block did catch the error and showed your message instead.

The last (but of course not the least) feature to help with structuring your code is the ability to put some often used functionality (such as utility functions) to a [module](https://technet.microsoft.com/en-us/library/dd878324(v=vs.85).aspx). Which is simply a PoSh file with a different extension (.psm1). You can easily import it to your script:

Import-Module "c:\MyPowerShellModules\MyPowerShellModule1.psm1" –DisableNameChecking

Once a module is loaded, it is in its own space and its functions will not be able to access your script variables. Even the ones that are explicitly set as global. The same is true about the script system variables (like $ErrorActionPreference above). They must be set inside the module to affect the module function behaviour.

Now, let’s have a look at some useful cmdlets. We already know how to scan the file system (Get-ChildItem). What if we need to check whether a path exists, and if it doesn’t,create one? Easy:

$path = "C:\MyFolder"

$found = Test-Path -Path $path

if(!$found)

{

New-Item -Path $path -ItemType directory

}

Test-Path $path

As DBAs we surely would like to run some SQL queries. So, here we go:

$ServerName = ".\LOCAL2016"

$DatabaseName = "master"

$Query = @"

select f1=11, f2=21 union select f1=21, f2=22

select f1='aa', f2='bb', CreatedOn='2017-01-01 01:01:01'

"@

$rows = Invoke-Sqlcmd -ServerInstance $ServerName -Database $DatabaseName -Query $Query -QueryTimeout 65535

foreach($row in $rows)

{

Write-Host $row.GetType() $row["f1"] $row["f2"]

}

It’s interesting to note that the cmdlet returns a collection of DataRow objects. Unfortunately, if you are returning results of multiple select statements it’s quite difficult to tell them apart. With introduction of SQL Server 2016, this has been improved and a new parameter, OutputAs, has been added. Now you can choose how to return your data: as a collection of DataRows, DataTables or a DataSet (the latter is preferable as it will return empty tables as well).

This cmdlet is not the only way to access your data. Remember that New-Object cmdlet? You can use it to work with your data in the .Net style. That is by creating all those .Net data access objects. To illustrate that let’s create something useful, for instance, a function to copy data from one server to another. First goes the function header:

function Copy-Data

{

Param([Parameter(Mandatory=$true)] [string] $SrcServer,

[Parameter(Mandatory=$true)] [string] $DstServer,

[Parameter(Mandatory=$true)] [string] $SrcDatabase,

[Parameter(Mandatory=$true)] [string] $DstDatabase,

[Parameter(Mandatory=$true)] [string] $CmdText,

[Parameter(Mandatory=$true)] [string] $DstTable,

[Parameter(Mandatory=$false)] [int] $BatchSize = 1000,

[Parameter(Mandatory=$false)] [int] $EstimatedCount = -1

)

Now, we create and use those .Net objects as we’d do that in .Net:

Try

{

$SrcConnStr = "Data Source=$SrcServer;Initial Catalog=$SrcDatabase;Integrated Security=SSPI"

$DstConnStr = "Data Source=$DstServer;Initial Catalog=$DstDatabase;Integrated Security=SSPI"

$SrcConn = New-Object System.Data.SqlClient.SQLConnection($SrcConnStr)

$SqlCommand = New-Object system.Data.SqlClient.SqlCommand($CmdText, $SrcConn)

$SrcConn.Open()

[System.Data.SqlClient.SqlDataReader] $SqlReader = $SqlCommand.ExecuteReader()

$bulkCopy = New-Object Data.SqlClient.SqlBulkCopy($DstConnStr, [System.Data.SqlClient.SqlBulkCopyOptions]::KeepIdentity)

$bulkCopy.DestinationTableName = $DstTable

$bulkCopy.NotifyAfter = $BatchSize

$bulkCopy.BatchSize = $BatchSize

}

The next section is to show the operation progress. It’s not a part of the actual data copying and can be skipped. Note the use of an event handler (the expression in curly brackets assigned to $Action variable). The handler gets called every time when NotifyAfter number of records are loaded. It uses Write-Progress cmdlet to render the progress. A variable $Sender (inside the handler) is an [auto-variable](https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/about/about_automatic_variables), something that PoSh creates and populates for you (in a context).

if($EstimatedCount -ne -1)

{

$global:totalCount = $EstimatedCount

$Action = {$rowsCopied += $Sender.NotifyAfter; Write-Progress -Activity "Row Copied:" -Status $rowsCopied -PercentComplete ($rowsCopied/$global:totalCount \*100)}

}

else

{

$Action = {$rowsCopied += $Sender.NotifyAfter; Write-Progress -Activity "Row Copied:" -Status $rowsCopied -PercentComplete -1}

}

Register-ObjectEvent -InputObject $bulkCopy -EventName "SqlRowsCopied" -Action $Action

And the copy statement itself with some error handling and finalizing:

$bulkCopy.WriteToServer($sqlReader)

}

Catch [System.Exception]

{

Write-Host $\_

}

Finally

{

$SqlReader.close()

$SrcConn.Close()

$SrcConn.Dispose()

$bulkCopy.Close()

}

}

Now, we just need to call it:

$CmdText = "select id, name from test1"

Copy-Data -SrcServer ".\LOCAL2016" `

-DstServer ".\LOCAL2016" `

-SrcDatabase "Test1" `

-DstDatabase "Test2" `

-CmdText $CmdText `

-DstTable "test2" `

-EstimatedCount 10000

Noticed the use of the “grave accent”? Here it’s a PoSh continuation operator. Very useful in organizing long expressions that cannot be made multi-line easily (like cmdlet calls).

So far so good, and being a DBA you surely want to know how to control your server from a PoSh script. PoSh has [a rich collection](https://docs.microsoft.com/en-us/powershell/sqlserver/sqlserver/vlatest/sqlserver) of cmdlets for that but I’d like to show how to use Server Management Objects (SMO) directly from your script.

Have you ever found yourself in a need to compare a set of objects in two different databases (let’s say production vs. development)? Quite often (especially if your databases are complex ones)? And you don’t have [SQL Compare](http://www.red-gate.com/products/sql-development/sql-compare/) and don’t want to go through the hassle of creating a full database compare project in [MS VS](https://docs.microsoft.com/en-us/visualstudio/install/install-visual-studio)? Let’s write a script to help us with this little task. So, how are we going to do that? We will create an instance of Microsoft.SqlServer.Management.Smo.Server through which we will gain an access to our objects and script them with an instance of Microsoft.SqlServer.Management.Smo.Scripter  into two folders. After that we will call “Beyond Compare 4” ([you can download it from here](https://www.scootersoftware.com/download.php)), a command line utility to compare the scripts (you can use any other comparison tool of your choice of course).

First, we declare script parameters:

Param(

[Parameter(Mandatory=$true)] [string] $SourceServer,

[Parameter(Mandatory=$true)] [string] $SourceDatabase,

[Parameter(Mandatory=$true)] [string] $TargetServer,

[Parameter(Mandatory=$true)] [string] $TargetDatabase,

[Parameter(Mandatory=$false)] [string] $ObjectFile = "objects.txt"

)

Where $ObjectFile is the name of the file which we store our object list in. The object names should be qualified with schema names, no square brackets or quotes (like dbo.MyTable).

Second, we add a simple function that merely writes to a file. Yes, we have the pre-built one for this but we need some extra formatting details which is better to control in one place:

function Write-Line

{

Param( [Parameter(Mandatory=$true)] [string] $FileName,

[Parameter(Mandatory=$true)] [AllowEmptyString()] [string] $Text,

[Parameter(Mandatory=$false)] [switch] $Overwrite = $false

)

if($Overwrite)

{

Write-Output $null | Out-File $FileName -Width 512

}

else

{

Write-Output $Text | Out-File $FileName -Width 512 -Append -NoClobber

}

}

Next, we notice that we have to extract metadata from both the source and the destination and it pretty much is the same routine. Let’s put it into a function. Here goes the function header:

function Generate-Script

{

Param(

[Parameter(Mandatory=$true)] [string] $ServerName,

[Parameter(Mandatory=$true)] [string] $DatabaseName,

[Parameter(Mandatory=$true)] [hashtable] $ObjectList,

[Parameter(Mandatory=$true)] [string] $Folder,

[Parameter(Mandatory=$true)] [string] $TimeStamp,

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

)

In the parameter list $Folder is obviously an output folder, $TimeStamp will be used as a file name prefix to differentiate current comparison “project” from the others and $FileNamePattern is a pattern according to which the script file name will be generated.

Now, we create our SMO Server and Scripter (remember that powerful New-Object thing?) and select the required database:

$Server = New-Object "Microsoft.SqlServer.Management.Smo.Server" $ServerName

$Scripter = New-Object "Microsoft.SqlServer.Management.Smo.Scripter" $Server

$db = $Server.Databases.Where( {$\_.Name -eq $DatabaseName})

if($db.Count -eq 0)

{

Write-Host "Database $DatabaseName not found on Server $ServerName!"

return

}

Choose what kind of objects we want:

$Objects = $db.Tables

$Objects += $db.Views

$Objects += $db.StoredProcedures

$Objects += $db.UserDefinedFunctions

Set some scripting options ([find more about them here](https://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.scriptingoptions.aspx)):

$Scripter.Options.NoFileGroup = $true

$Scripter.Options.NoCommandTerminator = $false

$Scripter.Options.NoCollation = $true

$Scripter.Options.IncludeHeaders = $false

$Scripter.Options.ScriptBatchTerminator = $true

And generate the script itself:

$eol = "`r`n" +"GO" +"`r`n"

foreach($obj in $Objects)

{

$tname = $obj.Schema +"." +$obj.Name

if($ObjectList.ContainsKey($tname))

{

$scriptcollection = $Scripter.Script($obj)

$script = $scriptcollection -join $eol

$script += $eol

$xname = $tname.Replace(".", "\_")

$filename = $FileNamePattern -f $Folder, $TimeStamp, $xname

Write-Line -FileName $filename -Text $script -Overwrite

}

}

}

Note the use of the script separator “GO” in the above snippet. As the Scripter returns a collection of strings each being an individual DDL or SET statement, it’s up to us to take care about their separation.

The function is done; we are halfway there. The next line in our script file deserves some explanation:

Import-Module "C:\Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\sqlps" –DisableNameChecking

The SQL Server related functionality is not a part of any default PoSh package. That said, you would have to import the required package (sqlps) in order to use it. Starting from version 3, PoSh will import it for you once you call any cmdlet from the package. But only if you call a cmdlet. If you want to use a type (like Microsoft.SqlServer.Management.Smo.Server) you are still on your own. Re-open PowerShell ISE and run the following snippet:

$ServerName = ".\LOCAL2016"

$Server = New-Object "Microsoft.SqlServer.Management.Smo.Server" $ServerName

PoSh will tell you that it cannot find the type!

In the Import-Module above, we could simply type “sqlps” instead of the full path, but there is a caveat. If you have more than one version of SQL Server installed on your machine, chances are that you have more than one version of that package and PoSh will import all of them. And that should be OK, but unfortunately sometimes it causes some odd errors. I, for instance, ran into type incompatibility error when tried to pass my collection of “RelocateFile” type objects created with a help of “New-Object” to “Restore-SqlServer” cmdlet. So, to avoid the gamble – directly load the latest and greatest version of the sqlps package. Next few lines in our script read the object list from a file and store it as a hashtable (for a quick access):

if(Test-Path $ObjectFile)

{

$content = Get-Content $ObjectFile

}

$list = @{}

foreach($o in $content)

{

$list.Add($o, "")

}

The next snippet checks whether output folders exist and creates them if they do not:

$SourceFolder = "Source"

$TargetFolder = "Target"

if(-not (Test-Path $SourceFolder))

{

New-Item -Path $SourceFolder -ItemType directory | Out-Null

}

if(-not (Test-Path $TargetFolder))

{

New-Item -Path $TargetFolder -ItemType directory | Out-Null

}

Now, we call the function we created earlier for the source and the target:

$OutputFileName = "{0}\{1}\_{2}.sql"

$TimeStamp = Get-Date -Format "yyyyMMddHHmmss"

Generate-Script -ServerName $SourceServer -DatabaseName $SourceDatabase -ObjectList $list -Folder $SourceFolder -TimeStamp $TimeStamp -FileNamePattern $OutputFileName

Generate-Script -ServerName $TargetServer -DatabaseName $TargetDatabase -ObjectList $list -Folder $TargetFolder -TimeStamp $TimeStamp -FileNamePattern $OutputFileName

At this point, scripts have been generated and stored, and all we need to do is just to call the comparison utility of your choice. Though in PoSh there are [a few ways to run an executable](https://social.technet.microsoft.com/wiki/contents/articles/7703.powershell-running-executables.aspx), Start-Process cmdlet is probably the best:

$compare = "C:\Program Files\Beyond Compare 4\BCompare.exe"

$args = """{0}"" ""{1}"" /expendall" -f $SourceFolder, $TargetFolder

Start-Process -FilePath $compare -ArgumentList $args –NoNewWindow

Save your script and run it from the Windows Command Prompt (cmd) like follows:

powershell .\Compare.ps1 -SourceServer ".\LOCAL2016" -SourceDatabase "Test1" -TargetServer ".\LOCAL2016" -TargetDatabase "Test2" -ObjectFile "objects.txt"

All the above, of course, is by no mean a comprehensive guide. To be honest, it barely scratches the surface as PoSh is massive. But I do hope that it provided you a good introduction on which you can build further on your own.