PowerShell Fast Track

Hacks for Non-Coders

Vikas Sukhija

Apress®

PowerShell Fast Track

Hacks for Non-Coders

Vikas Sukhija

Apress®

PowerShell Fast Track: Hacks for Non-Coders

Vikas Sukhija
Waterloo, ON, Canada

ISBN-13 (pbk): 978-1-4842-7758-4 ISBN-13 (electronic): 978-1-4842-7759-1
https://doi.org/10.1007/978-1-4842-7759-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: James Markham
Coordinating Editor: Mark Powers
Copy Editor: Mary Behr

Cover designed by eStudioCalamar

Cover image by Joshua Cotten on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484277584. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Vikas Sukhija has over a decade of IT infrastructure experience with expertise in messaging, collaboration, and IT automation utilizing PowerShell, PowerApps , Power Automate, and other tools. He is currently working as a Global Director at Golden Five Consulting in Canada. He is also a blogger, architect, and Microsoft MVP. He is known by the name TechWizard. As an experienced professional, he assists small to large enterprises in architecting, implementing, and automating Microsoft 365 and Azure.

About the Technical Reviewer

Arun Sharma is a techno-strategic leader and carries deep experience in development consulting, cloud, AI, and the IoT space. He is currently associated with Fresh & Pure, an agritech startup, in the position of Director Program Management-Automation and AI. He has expertise in various technologies (Microsoft Azure, AWS, Ali Cloud), IoT, ML, microservices, bots, Dynamics 365, PowerPlatform, SAP Crystal Reports, DevOps, Docker, and containerization. He has more than 20 years of experience in a wide range of roles such as GM-Paytm, Delivery Manager at Microsoft, Product Manager at Icertis, Lead and Architect Associate at Infosys, Executive Trainer at Aptech, and Development Consultant at CMC. He has managed CXO-level relationships on strategic levels, sales, cloud consumption, consulting services, and adoption with medium- and large-size global customers.

Introduction

This small book is full of small scripts that can be used by system administrators to improve the efficiency of their day-to-day IT operations. *PowerShell Fast Track* is for experienced IT administrators who want to utilize scripting and implement IT automations. Just copy/paste code blocks from the book/links to make simple to complex scripts. You can consider it your own personal scripting cheat book, like the cheat codes gamers utilize to ace electronic games. What it really is, however, is a practical guide, because it will get you started in automating much of your work.

CHAPTER 1

PowerShell Basics

Let's start with basic elements and quickly review variables, loops, `if/else` statements, switches, and functions. These are the heart of any scripting language, and will assist you in creating simple to complex scripts.

I will not delve into PowerShell versions or what PowerShell is. (But here is an easy-to-understand definition without going into depth: PowerShell is a task automation solution made up of a command-line shell and a scripting language.) I also won't talk about what platforms it can be used on or get and set commands, because this is not an in-depth book for learning the language.

The intent of this book is to teach you how to create scripts without having a deep knowledge of under-the-hood elements. This is an approach I've used successfully with many students. They gradually became well-versed with the language.

Not everyone is from a programming background and not everyone is adept at creating code. However, by following the approach described in this book, you will be able to quickly create your own scripts and automate IT systems/processes.

Note All source code used in the book can be accessed by clicking the **Download Source Code** button located at `www.apress.com/9781484277584` (follow the listing numbers).

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_1

Variables and Printing

To begin, you need to understand the basics, which include variables and arrays. Every variable in PowerShell starts with a dollar sign ($), such as

```
$a = "1"
$b = "Vikas"
```

When you type `$a` and `$b`, values will be displayed, as shown in Figure 1-1.

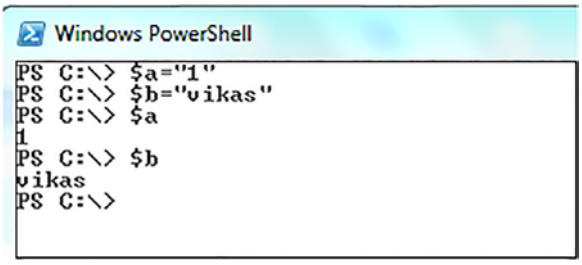


Figure 1-1. *Variables in PowerShell*

Now you can use `Write-host` to print this to the screen:

```
Input: PS C:\> Write-host $a
Output: 1
Input: PS C:\> Write-host $b
Output: vikas
```

Tip Make it a rule to use quotes when assigning values to variables, as shown above in the two examples.

Let's change the foreground color of what is being displayed by `Write-host`:

```
Input: PS C:\> Write-host $b -ForegroundColor Green
Output: vikas
Input: PS C:\> Write-host "processing ........."
-ForegroundColor Green
Output: processing .........
PS C:\>
```

Figure 1-2 shows the results.

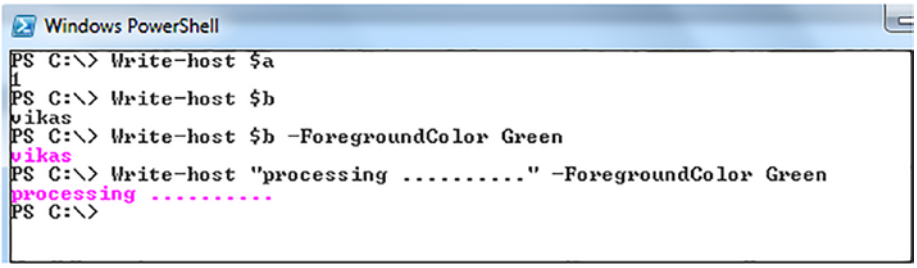


Figure 1-2. *Using the Write-host variable*

Let's illustrate arrays quickly. In PowerShell, arrays can be defined in the same way as variables. Here are some examples:

```
$b = "A","B","C","D","E"
```

If you define it as a variable and separate the elements by a comma, PowerShell automatically understands that it is an array, as shown in Figure 1-3.

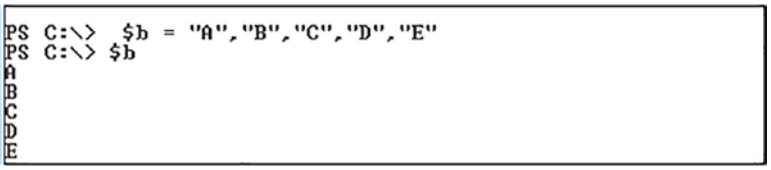


Figure 1-3. *Array illustration*

The proper way of defining an array is as follows because it is understood from the syntax itself that it is an array (as shown in Figure 1-4):

```
$c= @("server1","server2")
```

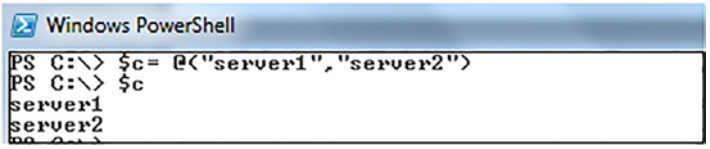


Figure 1-4. *Array syntax*

A dynamic array can be defined as `@()`. You will use this type in examples in this book to add elements in the array and generate reports:

```
$d = @()
```

If/Else Switch

`If else` is condition-based processing. It is the basis of any scripting language. If some condition is true, you need to process something; otherwise, process some other thing.

Listing 1-1 shows two examples. First, you define a variable value as 10 and then you use the conditional operators and `if else` statements to check if it's greater than 9 or if it's less than 9. Based on the result, you use `Write-host` to print it to screen as shown in Figure 1-5.

Note that `-gt` means greater than and `-lt` means less than. Do not worry; I will quickly go through them in the next subsection.

Listing 1-1. Example Code for Greater Than Operator Usage in If/Else

```
[int]$a= "10"
```

```
if($a -gt "9")
{
write-host "True" -foregroundcolor Green
}else {
Write-host "False" -foregroundcolor Red
}
```

Yes, [int] that means integer. If you use a prefix before the variable, it means you have exclusively defined it as an integer. Defining it is always better but if you don't, PowerShell is intelligent enough to do it implicitly. These are called datatypes, and they include [string], [char], [int], [array], etc.

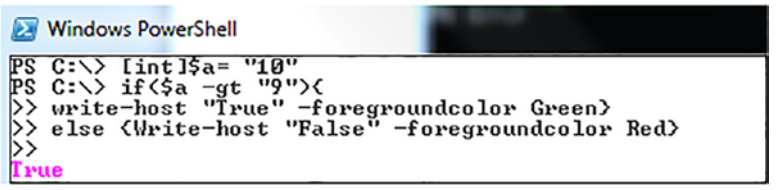


Figure 1-5. *Showing -gt usage in if/else*

Listing 1-2 and Figure 1-6 show a less than operator usage snippet.

Listing 1-2. Example Code for the Less Than Operator Usage in If/Else

```
[int]$a= "10"

if($a -lt "9"){
write-host "True" -foregroundcolor Green
}else {
Write-host "False" -foregroundcolor Red
}
```

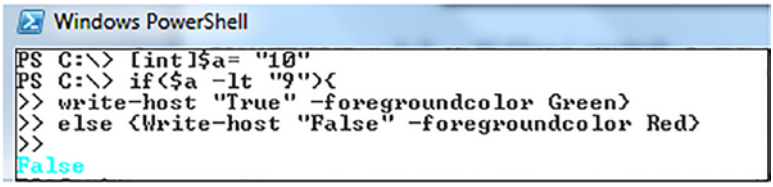


Figure 1-6. *Showing -lt usage in if/else*

Conditional/Logical Operators

Below is a list of conditional/logical operators that you will use in your everyday scripts. Without them, many scripting operations would not be possible. They will always be used in comparison `if else` statements as shown in the above parent section.

`-eq` Equal

`-ne` Not equal

`-ge` Greater than or equal

`-gt` Greater than

`-lt` Less than

`-le` Less than or equal

`-like` Wildcard comparison

`-notlike` Wildcard comparison

`-match` Regular expression comparison

`-notmatch` Regular expression comparison

`-replace` Replace operator

`-contains` Containment operator

`-notcontains` Containment operator

Logical Operators

-and Logical AND

-or Logical OR

-not Logical NOT

! Logical NOT

Logical operators are used when you want to combine the conditions. Let's update the above example to the code shown in Listing 1-3. You will print true if the value of variable $a is less than 9 or equals to 10. Here you have combined two conditions. Since it is the OR operator, TRUE will be returned if one of them matches. Here the second condition, `$a -eq "10"`, matches if a value is equal to 10. See the results in Figure 1-7.

Listing 1-3. Example Code Showing Logical -or Operator

```
[int]$a= "10"

If(($a -lt "9") -or ($a -eq "10")){
write-host "True" -foregroundcolor Green
}else {
Write-host "False" -foregroundcolor Red}
```

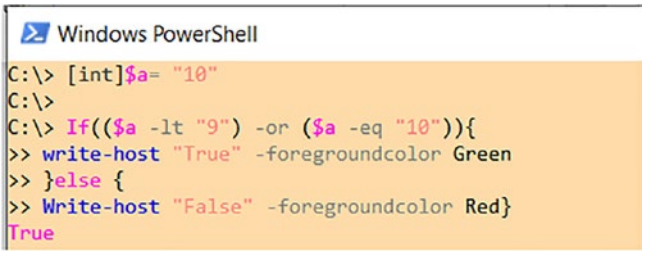


Figure 1-7. *Showing logical -or operator*

If you use the AND operator, then both conditions should match if you want to return TRUE, which will not happen in the above case. See Listing 1-4 and Figure 1-8.

Listing 1-4. Code Showing Logical -and Operator

```
[int]$a= "10"

If(($a -lt "9") -and ($a -eq "10")){
write-host "True" -foregroundcolor Green
}else {
Write-host "False" -foregroundcolor Red}
```

Figure 1-8. *Showing logical -and operator*

Not is for negation. I generally don't use it often. In my experience, it causes human error if you are in hurry and do not think it through enough.

Loops

There are two main loops in any scripting language and that is true for PowerShell as well. There are others but they are all variants of these two.

For Loop and While Loop

For Loop

There are three iterations of `for` loops in PowerShell:

- `foreach`
- `foreach-object`
- `for`

Let's differentiate between the three `for` loops by looking at the examples.

`foreach`: You need to specify a `foreach $variable` in `$collection`: `foreach ($i in $x)`.

Note You combine the `if else` and comparison operators in Listing 1-5. You can see the results in Figure 1-9.

Listing 1-5. Code Showing a foreach Loop

```
$x=@("1","2","3",,"4")

foreach ($i in $x) {
      if ($i -lt 2) { write-host "$i is Green" -foregroundcolor
      Green
      }
         else{ write-host "$i is yellow" -foregroundcolor
         yellow
      }
 }
```

Figure 1-9. *Showing a foreach loop*

`foreach-object`**:** You use a PIPE with the collection to achieve the same thing (see Listing 1-6 and Figure 1-10):

```
$x | foreach-object
```

Listing 1-6. Code Showing a foreach-object Loop

```
$x=@("1","2","3",,"4")

$x | foreach-object{
        if ($_ -lt 2) { write-host "$_ is Green"
        -foregroundcolor Green
}
        else{ write-host "$_ is yellow" -foregroundcolor yellow
}
    }
```

Figure 1-10. *Showing a foreach-object loop*

for: This is the one you will remember from your school days. I have not used it much and see less usage across the community. See the code in Listing 1-7 and the results in Figure 1-11.

Listing 1-7. Code Showing a for Loop

```
for($x=1; $x -le 5; $x++){
    if($x -lt 2){write-host "$x is Green" -foregroundcolor
    Green
    }
    else{ write-host "$x is yellow" -foregroundcolor yellow
    }
 }
```

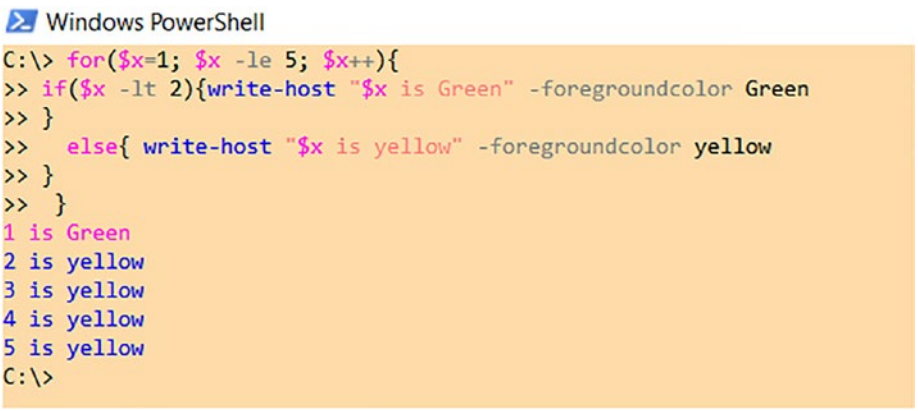


Figure 1-11. *Showing a for loop*

While Loop

The `while` loop is different because it lasts until the condition is true. Let's go through some examples to get more clarity.

The `while` loop also has two iterations:

- `do-while`
- `while`

For `do-while` you do something until some condition is met. In Listing 1-8, variable `x = 0` and inside the variable you increment its value until it is not equal to 4. See Figure 1-12 for the result.

Note You are doing the thing first and matching the condition later.

Listing 1-8. Code Showing a do-while Loop

```
$x= 0

Do {$x++
```

```
if($x -lt 2){write-host "$x is Green" -foregroundcolor Green
             }
            else{ write-host "$x is yellow" -foregroundcolor
            yellow
}
 }while($x -ne 4)
```

Figure 1-12. *Showing a do-while loop*

For `while, you are also` doing something until some condition is met. In Listing 1-9, variable `x = 0` and inside the variable you increment its value until it is not equal to 4.

Note You are checking first and doing the thing after that.

The main difference between the two, as you can see, is the `while` loop (an example of which is shown in Listing 1-9) checks the condition before the loop (iteration) but `do-while` does the checks after the execution. See Figure 1-13 for the result.

Listing 1-9. Code Showing the while Loop

```
$x= 0

while($x -ne 4) {$x++
if($x -lt 2){write-host "$x is Green" -foregroundcolor Green
       }
       else{ write-host "$x is yellow" -foregroundcolor yellow
      }
}
```

Figure 1-13. *Showing a while loop*

Functions

Functions are entities that, once defined, can be called anywhere in the script. Using functions avoids repetitive, lengthy code. Here's a glimpse for better understanding. I will not go in any details about parametrization and advanced functionalities of a function as my main motive here is a little bit of understanding and combining the code together to get the work done. In Listing 1-10, you create an Add function to add two numbers. The result is shown in Figure 1-14.

Listing 1-10. Example Code Showing an Add Function of Two Numbers

```
Function Add ($a1, $b1)
{
$a1 + $b1
}
```

Add 5 6 # Call function

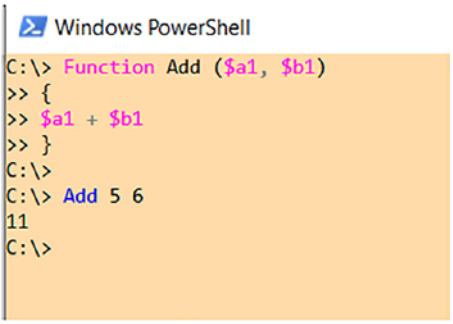


Figure 1-14. *Showing an Add function of two numbers*

Similarly, you can create this for three or more numbers. See Listing 1-11 and Figure 1-15.

Listing 1-11. Example Code Showing an Add Function of Three Numbers

```
Function Add ($a1, $b1, $c1)
{
$a1 + $b1 +$c1
}
```

Add 5 6 9 # Call function

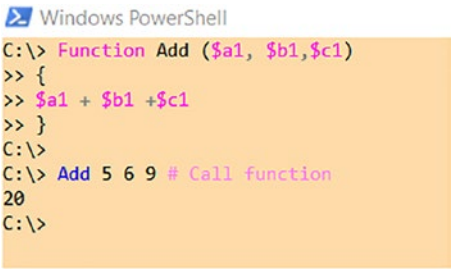


Figure 1-15. *Showing an Add function of three numbers*

Summary

In this chapter, you learned about PowerShell basics such as variables, arrays, if/else statements, and loops, which are the building blocks for creating powerful scripts in a production environment. These basics will be utilized further in the following chapters.

CHAPTER 2

Date and Logs

To create scripts, it is essential to understand how to use the date and time cmdlet to timestamp different types of operations, such as creating a time-stamped log file. I will share a cheat function that you can utilize inside your scripts to create a time-stamped log as well as time-stamped entries inside the script.

Let's go through some date and time illustrations before utilizing the `Write-Log` function that I have already created for you. (This is the first actual cheat code that you will utilize in your scripts!)

The `get-date` command provides you with the current date and time, as shown in Figure 2-1.

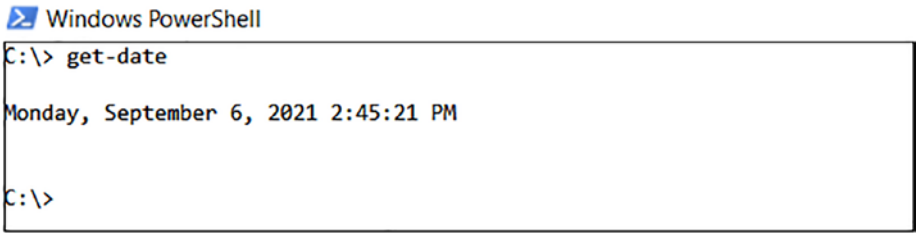


Figure 2-1. *Showing the get-date cmdlet*

To format it in a manner that will allow it to be used in file names and other instances, the `format` keyword can be used as shown in Figure 2-2:

```
get-date -format d
```


V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_2

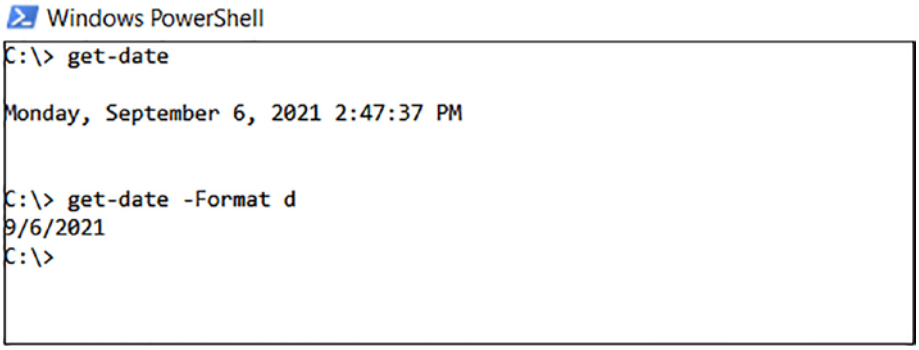


Figure 2-2. *Showing date formatting*

Listing 2-1 shows the date and time used in a file name.

Listing 2-1. Code for Date and Time Used in a File Name

```
$date = get-date -format d                  # formatting
$date = $date.ToString().Replace("/", "-")  # replace / with -
$time = get-date -format t    # only show time
$time = $time.ToString().Replace(":", "-") # replace : with -
$time = $time.ToString().Replace(" ", "")

$m = get-date
$month = $m.month  #getting month
$year = $m.year  #getting year
```

Examples: - (now gluing them all together)

```
#based on date
$log1 = ".\Processed\Logs" + "\" + "skipcsv_" + $date + "_.log"

#based on month and year
$log2 = ".\Processed\Logs" + "\" + "Created_" + $month +"_" +
$year +"_.log"
```

```
#based on date and time
$output1 = ".\" + "G_Testlog_" + $date + "_" + $time + "_.csv"
```

Note Always define the current working folder.

Date Manipulation

You saw that `get-date` can get you the current date and time. You can manipulate it according to the needs of your scripting solution. Here I will demonstrate briefly how to perform operations such as getting the first and last day of the month and getting a midnight date time stamp.

To get the first and last day of the month, you can use the code in Listing 2-2. See Figure 2-3 for the results.

Listing 2-2. Code for Fetching the First and Last Day of the Month

```
$date= Get-Date -Day 01
$lastday = ((Get-Date -day 01).AddMonths(1)).AddDays(-1)

$start = $date
$end  = $lastday
```

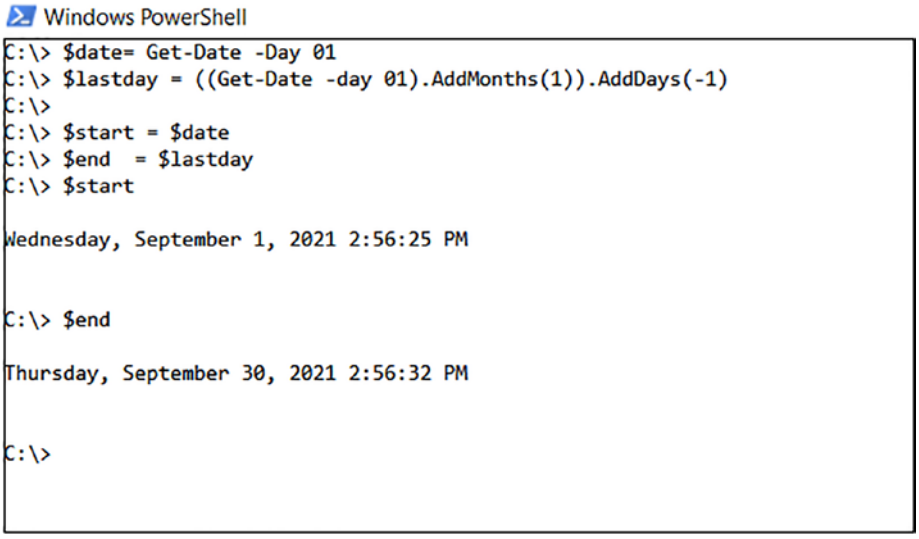


Figure 2-3. *Showing the first and last day of the month*

To get the midnight stamp, simply use this one-liner (and see Figure 2-4):

```
Get-Date -Hour 0 -Minute 0 -Second 0
```

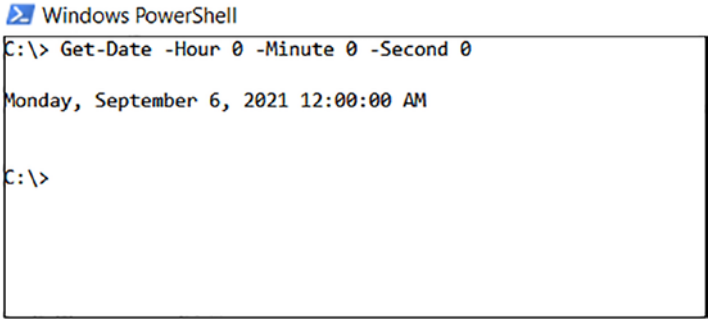


Figure 2-4. *Showing how to get the midnight date time stamp*

Creating Folders Based on a Date

There can be situations in the real world in which you want to create folders based on the current date, such as making a SharePoint configuration backup every day and placing it in the date stamp folder. Listing 2-3 shows the code that can be utilized to do this task and Figure 2-5 shows the results.

Listing 2-3. Code for Creating a Folder Structure Based on a Date

```
$Dname = ((get-date).AddDays(0).toString('yyyyMMdd')) #date
manipulation
$dirName = "ConfigBackup_$Dname"  #prefix for the folder

New-Item -Path c:\temp -Name $dirName -ItemType directory
```

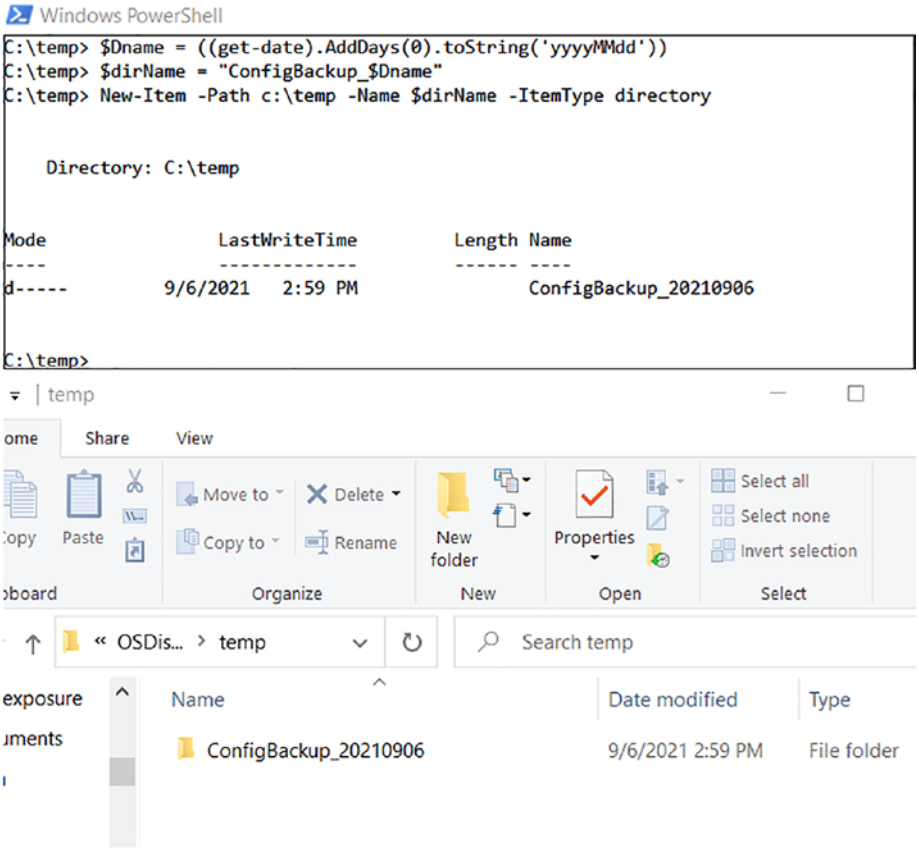


Figure 2-5. *Creating a folder structure based on a date*

Ready-Made Date and Log Functions

Here are three ready-made functions that you can copy and paste inside your scripts as per your requirements. Towards the end of this book, I will demonstrate how to create a complete script by using all the ready-made functions or code shared in this book.

Write-Log function: It uses another function named `New-FolderCreation`, which can be used separately if required. See Listing 2-4.

Listing 2-4. Code for Write-Log Function

```
function New-FolderCreation
{
  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $true)]
    [string]$foldername
  )
  $logpath  = (Get-Location).path + "\" + "$foldername"
  $testlogpath = Test-Path -Path $logpath
  if($testlogpath -eq $false)
  {
    #Start-ProgressBar -Title "Creating $foldername folder"
    -Timer 10
    $null = New-Item -Path (Get-Location).path -Name
    $foldername -Type directory
  }
}#New-FolderCreation

function Write-Log
{
  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $true,ParameterSetName = 'Create')]
    [array]$Name,
    [Parameter(Mandatory = $true,ParameterSetName = 'Create')]
    [string]$Ext,
    [Parameter(Mandatory = $true,ParameterSetName = 'Create')]
    [string]$folder,
```

```
    [Parameter(ParameterSetName = 'Create',Position = 0)]
    [switch]$Create,
    [Parameter(Mandatory = $true,ParameterSetName = 'Message')]
    [String]$message,
    [Parameter(Mandatory = $true,ParameterSetName = 'Message')]
    [String]$path,
    [Parameter(Mandatory = $false,ParameterSetName =
    'Message')]
    [ValidateSet('Information','Warning','Error')]
    [string]$Severity = 'Information',
    [Parameter(ParameterSetName = 'Message',Position = 0)]
    [Switch]$MSG
  )
  switch ($PsCmdlet.ParameterSetName) {
    "Create"
    {
      $log = @()
      $date1 = Get-Date -Format d
      $date1 = $date1.ToString().Replace("/", "-")
      $time = Get-Date -Format t

      $time = $time.ToString().Replace(":", "-")
      $time = $time.ToString().Replace(" ", "")
      New-FolderCreation -foldername $folder
      foreach ($n in $Name)
      {$log += (Get-Location).Path + "\" + $folder + "\" + $n +
      "_" + $date1 + "_" + $time + "_.$Ext"}
      return $log
    }
    "Message"
    {
      $date = Get-Date
```

```
        $concatmessage = "|$date" + "|   |" + $message +"|  |" +
        "$Severity|"
        switch($Severity){
          "Information"{Write-Host -Object $concatmessage
          -ForegroundColor Green}
          "Warning"{Write-Host -Object $concatmessage
          -ForegroundColor Yellow}
          "Error"{Write-Host -Object $concatmessage
          -ForegroundColor Red}
        }

        Add-Content -Path $path -Value $concatmessage
      }
    }
} #Function Write-Log
```

To create a log file, you can simply use it as below (it will auto-create the folders):

```
$log = Write-Log -Name "Name-Log" -folder "logs" -Ext "log"
```

To create a CSV file for report purposes, you can use it like so:

```
$Report1 = Write-Log -Name "MAM-Report" -folder "Report" -Ext "csv"
```

To write the information to a log file, you can use

```
Write-log -Message "Connect to Intune" -path $log
```

To write a warning to a log file, you can use

```
Write-log -Message "Connect to Intune" -path $log -Severity
Warning
```

To write an error to a log file, you can use

```
Write-Log -Message "Error loading Modules" -path $log -Severity
Error
```

Figure 2-6 shows the `Write-Log` operation in the PowerShell console.

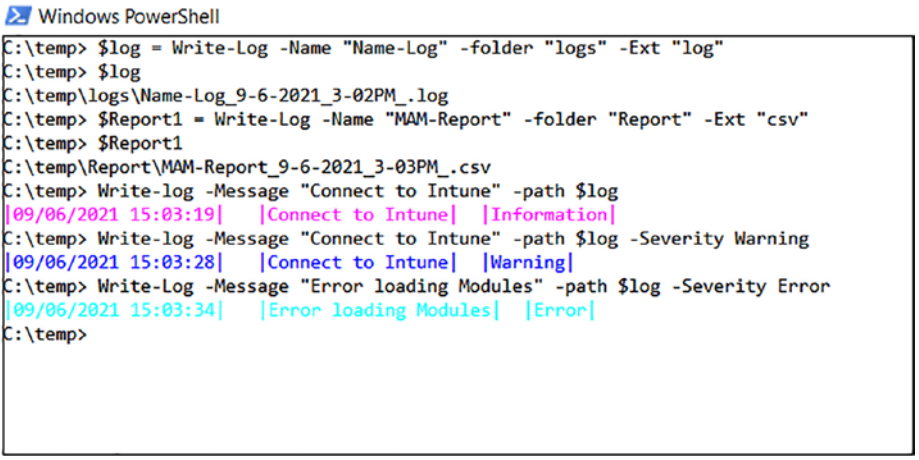


Figure 2-6. *Write-Log operation*

The log file is created is under the `logs` folder and will create a structural log text as shown in Figure 2-7.

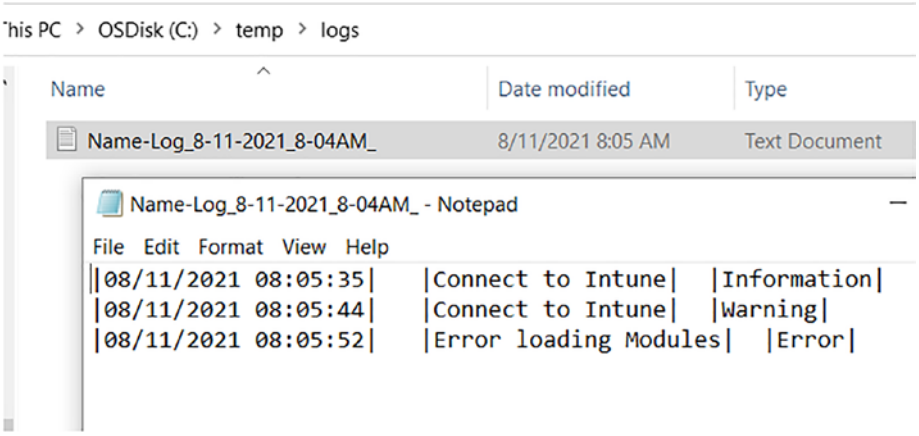


Figure 2-7. *Log file created after using the Write-Log function*

Set-Recyclelogs function: This will delete the files based on a number of days as input. As logs accumulate over time, there is a need to recycle them after a certain period to avoid filling up server drives. This is important for all scripts for which you have enabled logging. Use the code in Listing 2-5.

Listing 2-5. Code for the Set-Recyclelogs Function

```
function Set-Recyclelogs
{
  [CmdletBinding(
      SupportsShouldProcess = $true,
  ConfirmImpact = 'High')]
  param
  (
    [Parameter(Mandatory = $true,ParameterSetName = 'Local')]
    [string]$foldername,
    [Parameter(Mandatory = $true,ParameterSetName = 'Local')]
    [Parameter(Mandatory = $true,ParameterSetName = 'Path')]
    [Parameter(Mandatory = $true,ParameterSetName = 'Remote')]
```

```
    [int]$limit,

    [Parameter(ParameterSetName = 'Local',Position = 0)]
    [switch]$local,

    [Parameter(Mandatory = $true,ParameterSetName = 'Remote')]
    [string]$ComputerName,
    [Parameter(Mandatory = $true,ParameterSetName = 'Remote')]
    [string]$DriveName,
    [Parameter(Mandatory = $true,ParameterSetName = 'Remote')]
    [string]$folderpath,

    [Parameter(ParameterSetName = 'Remote',Position = 0)]
    [switch]$Remote,

    [Parameter(Mandatory = $true,ParameterSetName = 'Path')]
    [ValidateScript({
          if(-Not ($_ | Test-Path) ){throw "File or folder does
          not exist"}
          return $true
    })]
    [string]$folderlocation,

    [Parameter(ParameterSetName = 'Path',Position = 0)]
    [switch]$Path

  )

  switch ($PsCmdlet.ParameterSetName) {
    "Local"
    {
      $path1 = (Get-Location).path + "\" + "$foldername"
      if ($PsCmdlet.ShouldProcess($path1 , "Delete"))
```

```
  {
    Write-Host "Path Recycle - $path1 Limit - $limit"
    -ForegroundColor Green
    $limit1 = (Get-Date).AddDays(-"$limit") #for report
    recycling
    $getitems = Get-ChildItem -Path $path1 -recurse -file |
    Where-Object {$_.CreationTime -lt $limit1}
    ForEach($item in $getitems){
      Write-Verbose -Message "Deleting item $($item.
      FullName)"
      Remove-Item $item.FullName -Force
    }
  }
}
"Remote"
{
  $path1 = "\\" + $ComputerName + "\" + $DriveName + "$" +
  "\" + $folderpath
  if ($PsCmdlet.ShouldProcess($path1 , "Delete"))
  {
    Write-Host "Recycle Path - $path1 Limit - $limit"
    -ForegroundColor Green
    $limit1 = (Get-Date).AddDays(-"$limit") #for report
    recycling
    $getitems = Get-ChildItem -Path $path1 -recurse -file |
    Where-Object {$_.CreationTime -lt $limit1}
    ForEach($item in $getitems){
      Write-Verbose -Message "Deleting item $($item.FullName)"
      Remove-Item $item.FullName -Force
    }
  }
}
```

```
   "Path"
    {
      $path1 = $folderlocation
      if ($PsCmdlet.ShouldProcess($path1 , "Delete"))
      {
        Write-Host "Path Recycle - $path1 Limit - $limit"
        -ForegroundColor Green
        $limit1 = (Get-Date).AddDays(-"$limit") #for report
        recycling
        $getitems = Get-ChildItem -Path $path1 -recurse -file |
        Where-Object {$_.CreationTime -lt $limit1}
        ForEach($item in $getitems){
          Write-Verbose -Message "Deleting item $($item.FullName)"
           Remove-Item $item.FullName -Force
        }
      }
    }
  }

}# Set-Recycle logs
```

To recycle logs older than 10 days inside the logs folder in the current directory:

```
Set-Recyclelogs -foldername logs -limit 10
```

Use confirm:$false to avoid confirmation once you are sure that you want to delete the files:

```
Set-Recyclelogs -foldername logs -limit 10 -confirm:$false
```

Use verbose to check which files are getting deleted:

```
Set-Recyclelogs -foldername logs -limit 10 -confirm:$false -verbose
```

You can specifiy the path as well if your script is in another directory and you want to delete logs in another folder structure:

```
Set-Recyclelogs -folderlocation c:\temp\logs -limit 10
```

To recycle logs on a remote machine, use the following syntax:

```
Set-Recyclelogs -ComputerName testmachine -DriveName c
-folderpath data\logs -limit 10
```

Set-ProgressBar function: This function is just to show the progress bar when you want to pause for some time. See Listing 2-6 and Figure 2-8.

Listing 2-6. Code for Start-ProgressBar Function

```
function Start-ProgressBar
{
  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $true)]
    $Title,
    [Parameter(Mandatory = $true)]
    [int]$Timer
  )

  For ($i = 1; $i -le $Timer; $i++)
  {
    Start-Sleep -Seconds 1;
    Write-Progress -Activity $Title -Status "$i" -Percent
    Complete ($i /100 * 100)
  }
} #Function Start-ProgressBar
```

Start-ProgressBar -Title “Test timeout” -Timer 30

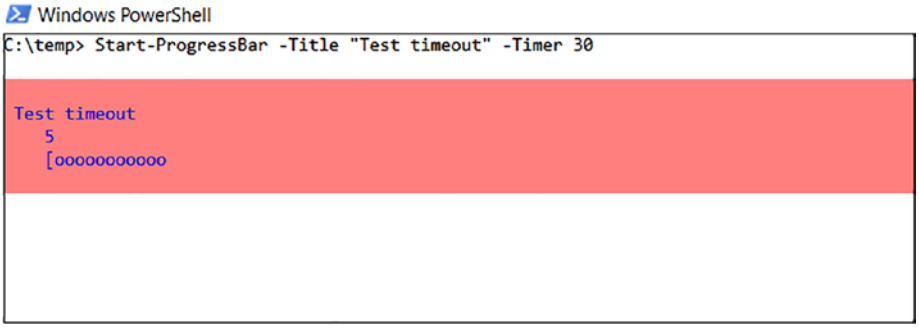


Figure 2-8. *Start-ProgressBar with a timer of 30 seconds*

You can use a simple timeout as well, which is built in (see Figure 2-9):

```
timeout 10
```

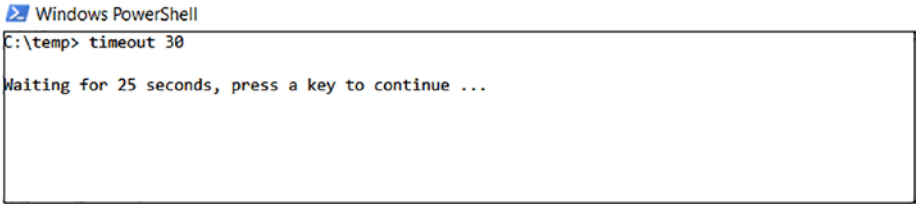


Figure 2-9. *Built-in timeout cmdlet*

Summary

In this chapter, you learned about the date and logs cmdlet and how to manipulate and use the format of dates to generate time-stamped folders and files. This technique will help you in different scenarios such as creating log files or log folders every day with a date/time stamp inserted into the file name.

CHAPTER 3

Input to Your Scripts

There are many situations in the practical system administration world in which you have to feed your scripts with inputs. Examples are reading a text file that has a list of users and adding them to a specific Active Directory group, or reading a CSV file that includes user attributes like phone number, title, and department, and updating these attributes in Active Directory. In this chapter, you will look at the different ways of feeding your scripts with different types of inputs.

Import-CSV

`Import-CSV` is the most-used method for providing a script with a CSV file to read and it is further used to perform bulk operations using loops.

To demonstrate, let's create a small CSV file (save it as `samplecsv.csv`) in the format shown in Figure 3-1 and then print the contents. See Listing 3-1 for the code.

A	B	C
User	email	title
Vikas Sukhija	svikas@techwizard.cloud	Blogger
Pradip Sukhija	sukhijap@techwizard.cloud	sysadmin

Figure 3-1. *Example CSV file*

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_3

Listing 3-1. Code for Import-CSV

```
$data = import-csv c:\temp\samplecsv.csv    #Import CSV in
                                             variable data
foreach ($i in $data) {
Write-host  $i.user -foregroundcolor green #printing column user
Write-host  $i.email -foregroundcolor yellow #printing column email
Write-host  $i.title -foregroundcolor magenta #printing column title
         }
```

Figure 3-2 shows the `Import-CSV` operation in PowerShell. If you are in the same folder, you can use dot source instead of the full path shown in screenshot:

```
$data = import-csv .\samplecsv.csv # .\ means current directory
```

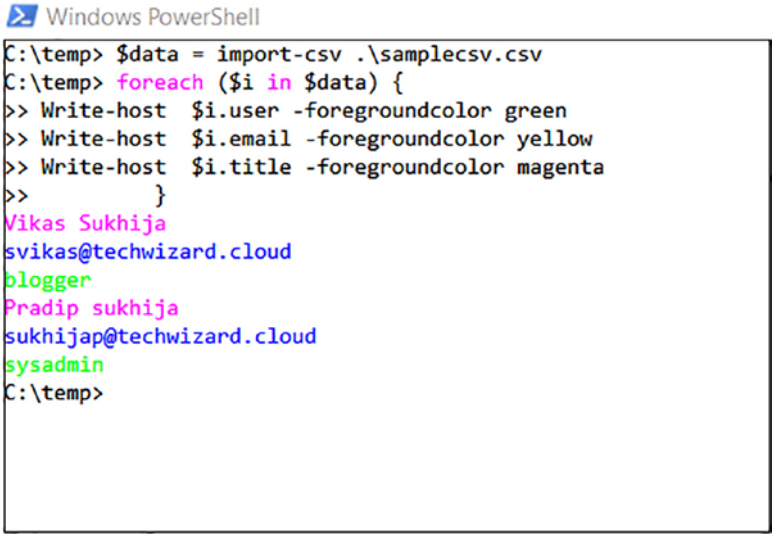


Figure 3-2. *Showing the Import-CSV operation by dot sourcing (.\)*

Importing from a Text File

There are scenarios in which you get data in a text file, such as a server list or a user list, one name at a time, and you want to perform a certain operation on the data.

Figure 3-3 shows the printing of each server from a file named `servers.txt` file onto the screen. (`Get-content` is the cmdlet used for reading text files; see Listing 3-2.)

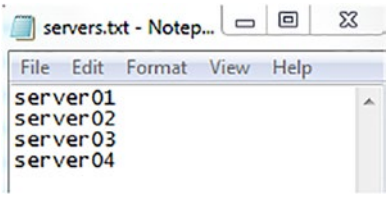


Figure 3-3. *Example text file contents*

Listing 3-2. Code for Reading from a Text File

```
$servers = Get-content .\servers.txt
$servers | foreach-object {
Write-host $_
}
```

Save the code in a `.ps1` file and run it or just paste it in the PowerShell console. See Figure 3-4.

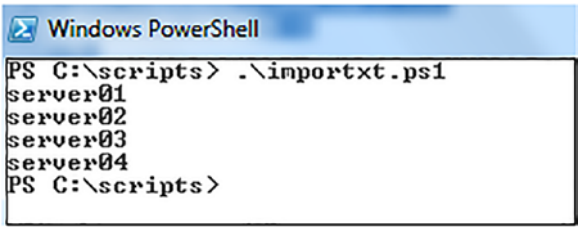


Figure 3-4. *Reading from a text file operation in PowerShell*

Input from an Array

You can do the same thing with array that you have done with text file. Say you have array of servers and you want to print it on the screen. See Listing 3-3.

Listing 3-3. Code for Reading from an Array and Printing It

```
$servers = @("server01","server02","server03","server04")
#array of servers
$servers | foreach-object {
Write-host $_ -foregroundcolor yellow
}
```

Running this script will show the results in Figure 3-5.

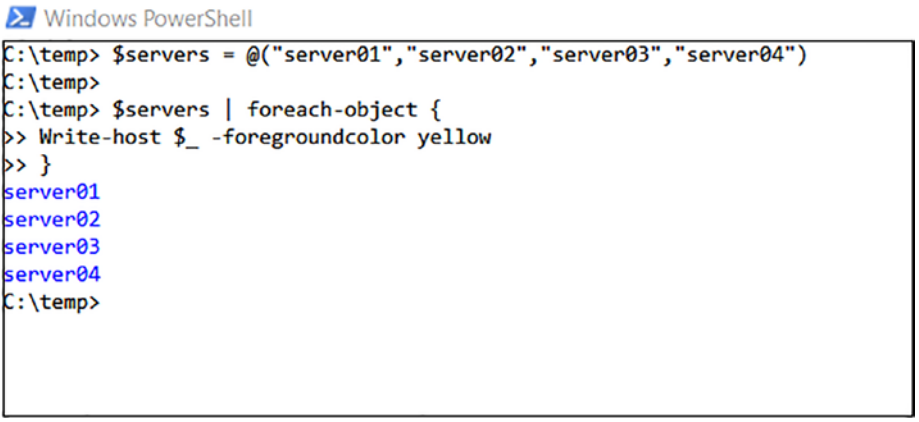


Figure 3-5. *Showing the printing of an array*

Summary

In this chapter, you learned how to feed scripts with input either through a text file, CSV file, or an array. There are advanced ways to input your scripts but the ways mentioned in this chapter are common and are used every day in the system administration world.

CHAPTER 4

Interactive Input

This chapter will provide examples where you can add interactive input to your scripts (i.e., the script will ask for input such as entering a password or another attribute such as the path of a CSV or text file).

Read-host

You have used `Write-host` for printing value to the screen. Now you can use `Read-host` to get values that are input by a user on the screen:

```
$x =Read-host "input your Name"
```

The value of your input is saved in an `$x` variable so you can use it in the script for further processing, as shown in Figure 4-1.

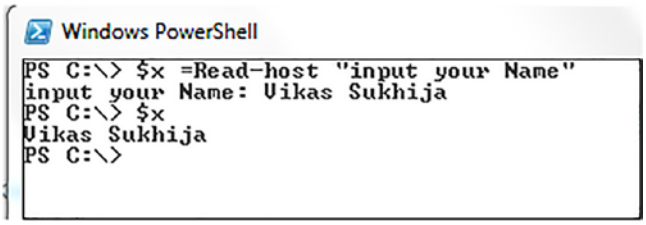


Figure 4-1. *Read-host operation*

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_4

Another option is to specifically use the `-prompt` parameter. There is no difference in how you use it. See Figure 4-2.

```
$Age =Read-host -prompt "input your Age."
```

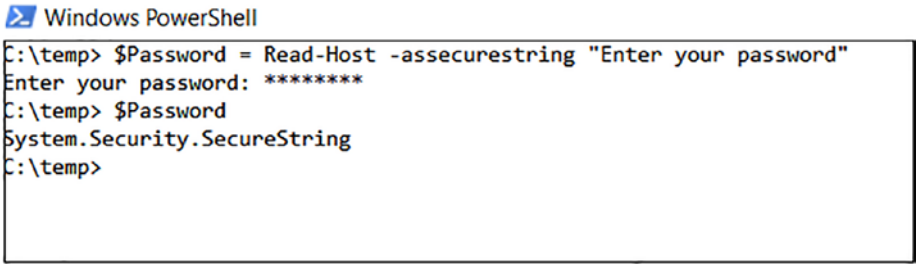


Figure 4-3. *Read-host operation for a password as an input*

-assecurestring is used when you want to input a password securely:

```
$Password = Read-Host -assecurestring "Enter your password"
```

Here as well the value of the password entered is saved in the `$Password` variable that you can use it in the script for further processing, such as authentication to some service like Office 365. See Figure 4-3.

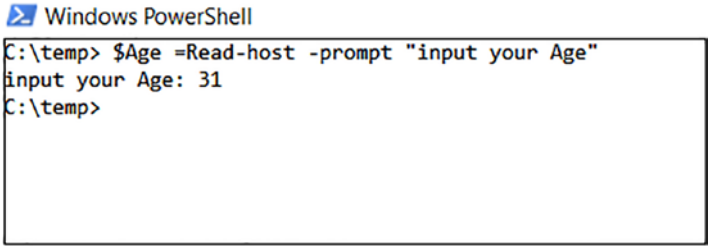


Figure 4-2. *Read-host operation specifying the -prompt*

Parameters

In a PowerShell command, functions are scripts that rely on parameters so that a user can either enter values or select options. I will briefly touch on basic parametrization. (Advanced parameters are outside the scope of this book.) See Listing 4-1.

Listing 4-1. Example Code Showing Use of Parameters

```
Param(
  [string]$firstname,
  [string]$lastname,
  [string]$title
)

Write-host "First Name: $firstname" -ForegroundColor Yellow
Write-host "Last Name: $lastname" -ForegroundColor Yellow
Write-host "Title: $Title" -ForegroundColor green
```

Save this as a `.ps1` file and run it as follows (and see Figure 4-4):

```
.\script.ps1 -firstname Vikas -lastname sukhija -title blogger
```

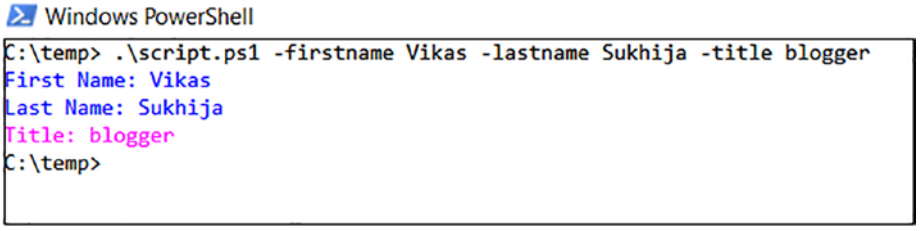


Figure 4-4. *Script execution with parameters*

Tip Make sure you define the parameters at the beginning of your script.

GUI Button

Here is a cheat code (function) in case you want interactive input in the form of a graphical user interface. I consider it as a fancy way to get input from the user. The button function in Listing 4-2 can take inputs from the Windows form that you can display on a screen or perform other desired operations in your script.

Listing 4-2. Code for Input from a GUI Button

```
function button ($title,$mailbx, $WF, $TF)
{
  ###################Load Assembly for creating form &
  button######
  [void][System.Reflection.Assembly]::LoadWithPartialName(
  "System.Windows.Forms")
  [void][System.Reflection.Assembly]::LoadWithPartialName(
  "Microsoft.VisualBasic")

  #####Define the form size & placement
  $form = New-Object "System.Windows.Forms.Form";
  $form.Width = 500;
  $form.Height = 150;
  $form.Text = $title;
  $form.StartPosition = [System.Windows.Forms.FormStartPosition
  ]::CenterScreen;

  ###############Define text label1
  $textLabel1 = New-Object "System.Windows.Forms.Label";
  $textLabel1.Left = 25;
  $textLabel1.Top = 15;

  $textLabel1.Text = $mailbx;
```

```
##############Define text label2

$textLabel2 = New-Object "System.Windows.Forms.Label";
$textLabel2.Left = 25;
$textLabel2.Top = 50;

$textLabel2.Text = $WF;

##############Define text label3

$textLabel3 = New-Object "System.Windows.Forms.Label";
$textLabel3.Left = 25;
$textLabel3.Top = 85;

$textLabel3.Text = $TF;

############Define text box1 for input
$textBox1 = New-Object "System.Windows.Forms.TextBox";
$textBox1.Left = 150;
$textBox1.Top = 10;
$textBox1.width = 200;

############Define text box2 for input

$textBox2 = New-Object "System.Windows.Forms.TextBox";
$textBox2.Left = 150;
$textBox2.Top = 50;
$textBox2.width = 200;

############Define text box3 for input

$textBox3 = New-Object "System.Windows.Forms.TextBox";
$textBox3.Left = 150;
$textBox3.Top = 90;
$textBox3.width = 200;
```

```
##############Define default values for the input boxes
$defaultValue = ""
$textBox1.Text = $defaultValue;
$textBox2.Text = $defaultValue;
$textBox3.Text = $defaultValue;

##############define OK button
$button = New-Object "System.Windows.Forms.Button";
$button.Left = 360;
$button.Top = 85;
$button.Width = 100;
$button.Text = "Ok";

############## This is when you have to close the form after
getting values
$eventHandler = [System.EventHandler]{
  $textBox1.Text;
  $textBox2.Text;
  $textBox3.Text;
  $form.Close();
};

$button.Add_Click($eventHandler) ;

##############Add controls to all the above objects defined
$form.Controls.Add($button);
$form.Controls.Add($textLabel1);
$form.Controls.Add($textLabel2);
$form.Controls.Add($textLabel3);
$form.Controls.Add($textBox1);
$form.Controls.Add($textBox2);
$form.Controls.Add($textBox3);
$ret = $form.ShowDialog();
```

```
  ##################return values

  return $textBox1.Text, $textBox2.Text, $textBox3.Text
} #button
```

Load this function into your script, and then you can perform the operations on the inputs as shown:

```
$return= button "Enter Folders" "Enter mailbox" "Working
Folder" "Target Folder"
```

You can choose different names per your requirements. See Figure 4-5.

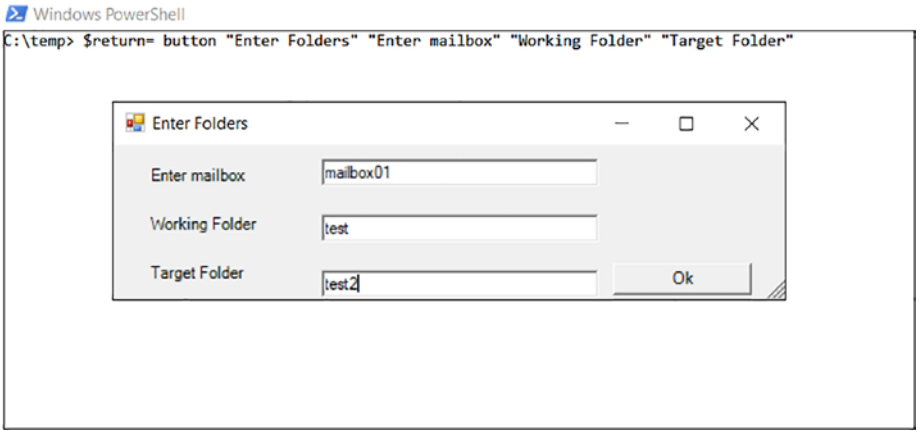


Figure 4-5. *GUI button input*

After you press the OK button, the `$return` variable contains all these values in the array (see Figure 4-6):

```
$return[0] → Enter mailbox value
$return[1] → Working folder value
$return[2] → Target Folder value
```

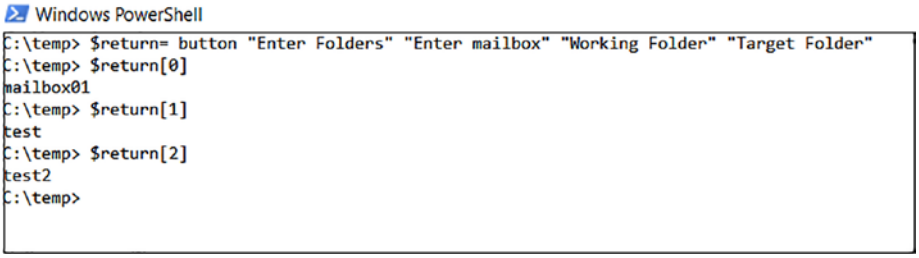


Figure 4-6. *Showing values returned from the user input*

You can also print to the screen in the same manner as shown previously (see Figure 4-7):

```
Write-host "Enter mailbox : $($return[0])" -ForegroundColor Yellow
Write-host "Working folder : $($return[1])" -ForegroundColor Yellow
Write-host "Target Folder : $($return[2])" -ForegroundColor green
```

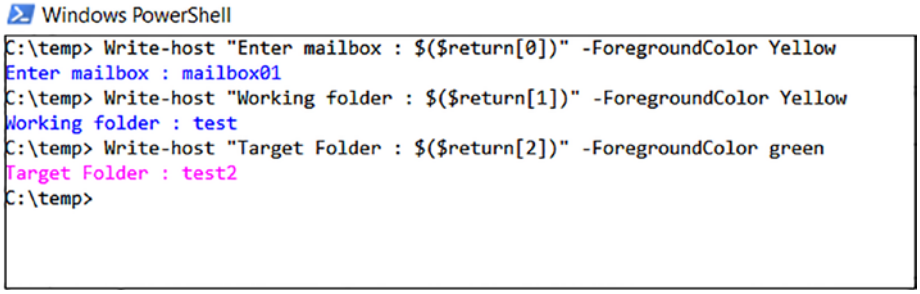


Figure 4-7. *Printing the values from the input using Write-host*

Prompt (Yes or No)

There are practical situations when as a system administrator you want to build a nice way to get a Yes/No response from the users. You can do this easily with PowerShell utilizing the cheat code in Listing 4-3.

Listing 4-3. Code for a Yes/No Operation

```
$overwrite = New-Object -comobject wscript.shell

$Answer = $overwrite.popup("Do you want to Overwrite AD
Attributes?",0,"Overwrite Attributes",4)

If ($Answer -eq 6) {Write-Host "you pressed Yes"
-ForegroundColor Green}
else{Write-Host "you pressed Yes" -ForegroundColor Red}
```

Copy and paste the code into the PowerShell console or save the script as a `.ps1` file and run it. See Figure 4-8.

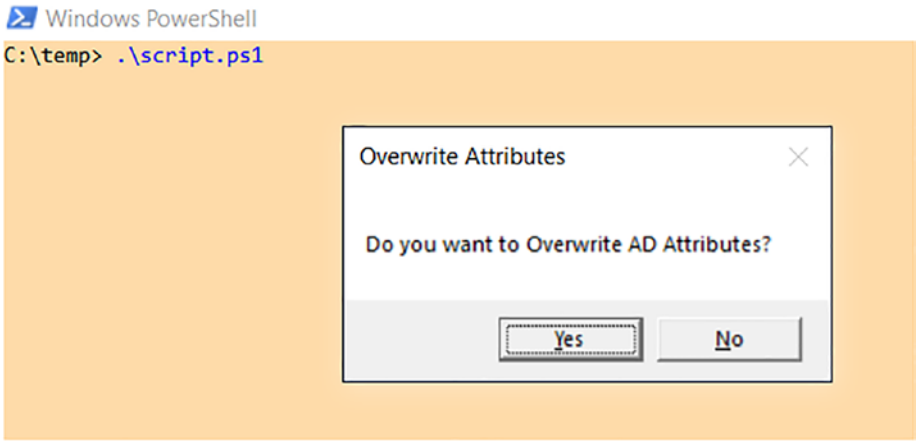


Figure 4-8. *Showing a Yes/No operation in PowerShell*

If you press Yes, you get the result shown in Figure 4-9.

Figure 4-9. *Showing the Yes operation*

If you press No, you get the result shown in Figure 4-10.

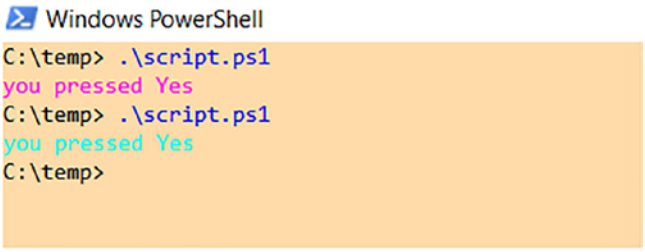


Figure 4-10. *Showing the No operation*

Instead of just `Write-host` you can perform different operations inside your script based on the response selected by the user.

Summary

In this chapter, you learned about interactive inputs. This strategy is utilized when you have to provide the script to an end user. The end user can just run the script. Questions that are coded by the scripter pop up interactively and the user can answer them and perform a meaningful job.

CHAPTER 5

Adding Snapins/ Modules

Microsoft and other third-party vendors have built PowerShell snapins or modules for their different products. To use the PowerShell cmdlets for these technologies, you have to either add the snapins or import the modules in your scripts.

PowerShell snapins are legacy products because nowadays everything comes as modules. You can consider a module as a battery that is required to run your scripts. Each module is a package that contains cmdlets, providers, functions, aliases, and more.

Although snapins are legacy, let's touch on them briefly, showing the products that use(d) them.

PowerShell Snapins

Exchange Sever itself is the better example to show for PowerShell snapins. Exchange 2007 and 2010 both used snapins.

To add an Exchange snapin to your scripts, you can use following code. (Examples of Exchange 2007 and 2010 snapins are in Listing 5-1 and Listing 5-2, respectively.) As these products are officially at end-of-support status, the usage might be rare (they're only applicable for organizations that have not upgraded yet).

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_5

Note Exchange management binaries should be installed first on the machine or the snapin will not work.

Listing 5-1. Code to Add the Exchange 2007 Management Shell

```
If ((Get-PSSnapin | where {$_.Name -match "Exchange.
Management"}) -eq $null)
{
       Add-PSSnapin Microsoft.Exchange.Management.PowerShell.
       Admin
}
```

Listing 5-2. Code to Add the Exchange 2010 Management Shell

```
If ((Get-PSSnapin | Where-Object { $_.Name -match "Microsoft.
Exchange.Management.PowerShell.E2010" }) -eq $null) {
    Add-PSSnapin Microsoft.Exchange.Management.PowerShell.E2010
}
```

After the snapin has been added to the session or the script, it can run the Exchange commands inside the window, as shown in Figure 5-1.

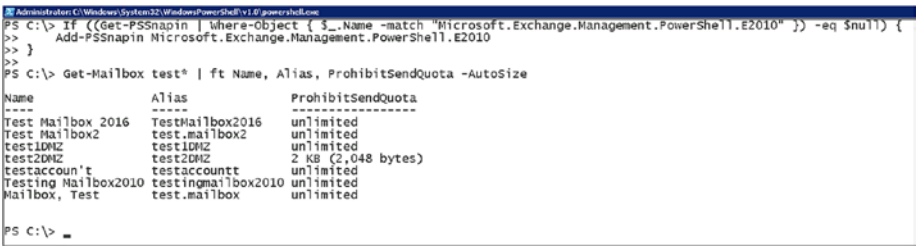


Figure 5-1. *Exchange Management Shell*

You can also use `get-pssnapin` in the PowerShell shell window to check which snapin is available.

In Figure 5-2, I want to take advantage of the Quest shell, so I use `get-pssnapin` in the Quest AD shell to find out the snapin name.

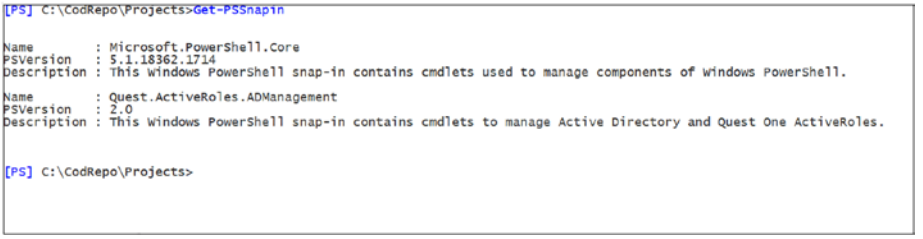


Figure 5-2. *Quest AD Management Shell*

Listing 5-3 shows how to add the snapin to the PowerShell script or session using the same technique as for the Exchange product.

Listing 5-3. Code to Add the Quest AD Management Shell

```
If ((Get-PSSnapin | where {$_.Name -match "Quest.ActiveRoles"})
-eq $null)
{
    Add-PSSnapin Quest.ActiveRoles.ADManagement
}
```

Note The above code to add a snapin first checks to see if the snapin already exists. If so, it does nothing. If not, it adds the required snapin.

Modules

Let's cover modules now as this is what you will deal with in your day-to-day work. Per Microsoft, "a module is a package that contains PowerShell members, such as cmdlets, providers, functions, workflows, variables, and aliases. The members of this package can be implemented in a PowerShell script, a compiled DLL, or a combination of both. These files are usually grouped together in a single directory."

For the PowerShell scripting language, we need modules to interact with different products, such as Azure, Office 365, Exchange Online, and so on.

PowerShell Gallery (`www.powershellgallery.com/`) is the de facto repository that contains almost all of the modules that anyone can utilize.

To install a module on your machine from PowerShell gallery, use this code:

```
Install-Module -Name AzureAD
```

Enter yes (as shown in Figure 5-3) when you receive the prompt to install the module.

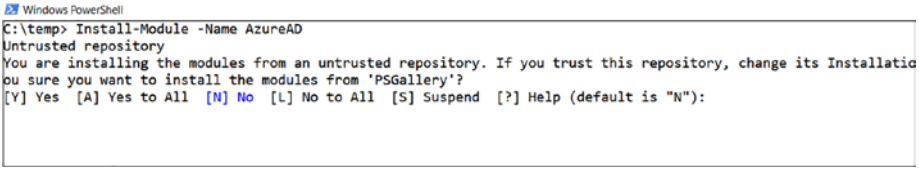


Figure 5-3. *Installing a module from PowerShell Gallery*

When you install the module on your machine it will get stored in `C:\Program Files\WindowsPowerShell\Modules` as depicted in Figure 5-4.

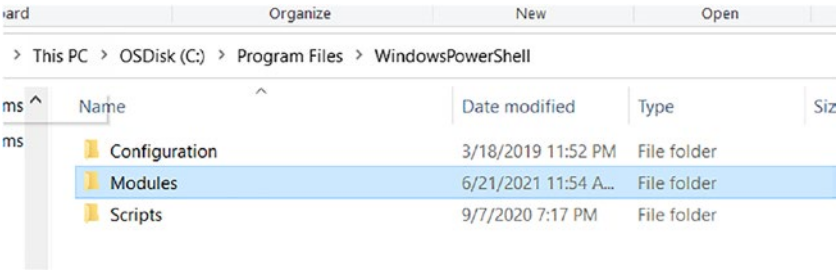


Figure 5-4. *Module path on the computer*

There may be a situation where a developer has developed a new version of the module and you want to update your machine with this newer version. Use one of the following commands to upgrade the existing module:

```
Update-Module -Name AzureAD
```

or

```
Install-Module -Name AzureAD -force (this will also upgrade the
module to latest version)
```

With `update` you can also update the module to a specific version:

```
Update-Module -Name AzureAD -RequiredVersion 1.0.1
```

Removing the module is a simple operation and can be done as shown in the following cmdlet:

```
Remove-Module AzureAD
```

After you have installed the module, which is a one-time task, and you want to utilize that module in your scripts, you can do so by using the `Import-Module` command.

Note After PowerShell V3, modules are loaded automatically when the first cmdlet from that module is run from the script.

You can still follow the practice of importing the modules in your script before running a command:

```
Import-Module AzureAD
```

To get all of the modules installed on your machine, you can use the following code (the results are shown in Figure 5-5):

```
Get-Module -ListAvailable
```

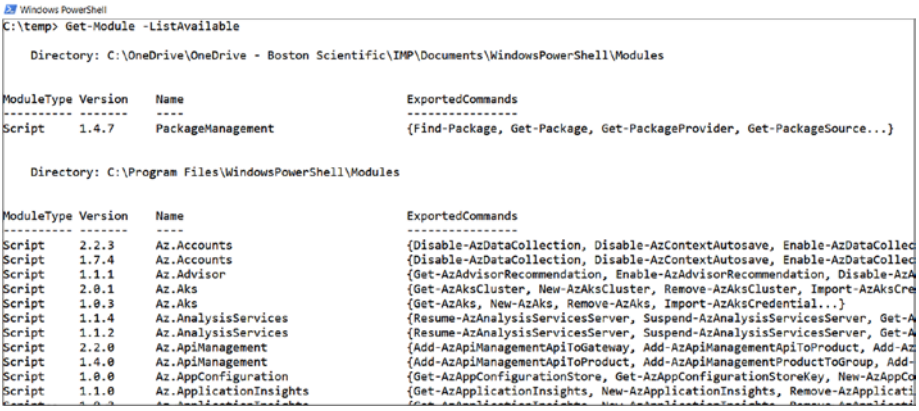


Figure 5-5. *Showing a list of available modules on the computer*

Cheat Module (vsadmin)

Since this book is about cheat codes to create complex scripts, here is a cheat module that is full of functions that you can utilize to do complex operations inside your scripts. I recently created this module for the community to help them with scripting. I update it to newer versions from time to time as features change for different products.

Module name: `vsadmin`

Installing the module (see Figure 5-6):

```
Install-Module -Name vsadmin
```

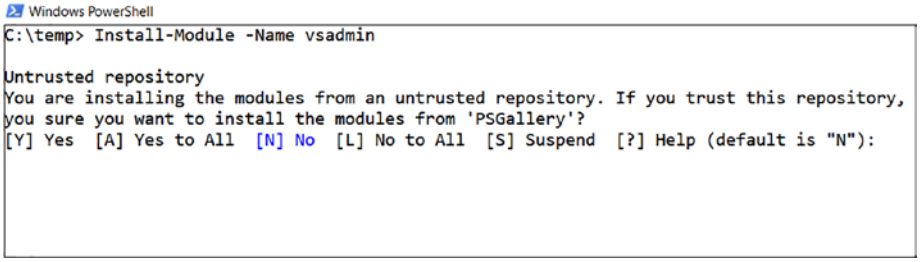


Figure 5-6. *Installing the vsadmin module*

Once installed, you will find files created inside your module's directory (`C:\Program Files\WindowsPowerShell\Modules`) as depicted in Figure 5-7.

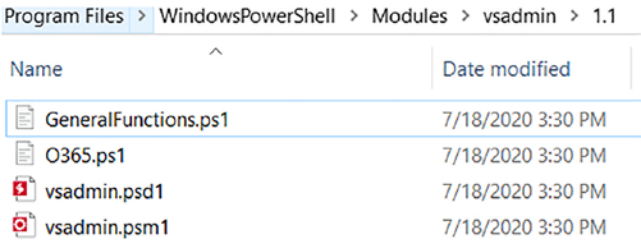


Figure 5-7. *Showing files inside the vsadmin module*

As stated, you can import the module in to the session using `import-module` like so:

```
import-module vsadmin
```

Figure 5-8 shows the commands that are available inside the `vsadmin` module. You can use the following command to check functions and cmdlets in any module:

```
Get-Command -Module vsadmin
```

Windows PowerShell

```
C:\temp> Get-Command -Module vsadmin

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Get-ADGroupMembersRecursive                        3.0        vsadmin
Function        Get-ADUserMemberOf                                 3.0        vsadmin
Function        Get-Auth                                           3.0        vsadmin
Function        Get-IniContent                                     3.0        vsadmin
Function        Group-Validate                                     3.0        vsadmin
Function        LaunchCOL                                          3.0        vsadmin
Function        LaunchEOL                                          3.0        vsadmin
Function        LaunchEXOnprem                                     3.0        vsadmin
Function        LaunchMSOL                                         3.0        vsadmin
Function        LaunchSOL                                          3.0        vsadmin
Function        LaunchSPO                                          3.0        vsadmin
Function        New-FolderCreation                                 3.0        vsadmin
Function        New-RandomPassword                                 3.0        vsadmin
Function        RemoveCOL                                          3.0        vsadmin
Function        RemoveEOL                                          3.0        vsadmin
Function        RemoveEXOnprem                                     3.0        vsadmin
Function        RemoveMSOL                                         3.0        vsadmin
Function        RemoveSOL                                          3.0        vsadmin
Function        RemoveSPO                                          3.0        vsadmin
Function        Save-CSV2Excel                                     3.0        vsadmin
Function        Save-EncryptedPassword                             3.0        vsadmin
Function        Send-Email                                         3.0        vsadmin
Function        Set-Recyclelogs                                    3.0        vsadmin
Function        Start-ProgressBar                                  3.0        vsadmin
Function        Test-vsadmin                                       3.0        vsadmin
Function        Write-Log                                          3.0        vsadmin


C:\temp>
```

Figure 5-8. *Available vsadmin module commands*

Let's explore Office 365 functions and then we will move on to other system admin functions that you can utilize daily.

- `LaunchEOL/RemoveEOL` (Exchange Online)
- `LaunchSOL/RemoveSOL` (Skype online)
- `LaunchSPO/RemoveSPO` (SharePoint online)
- `LaunchCOL/RemoveCOL` (Security and Compliance)
- `LaunchMSOL/RemoveMSOL` (MSonline Azure Active Directory)
- `LaunchEXOnprem/RemoveEXOnprem` (for on-premise Exchange Server)

Three of the Office 365 functions are prefixed so that they do not conflict with on-premise commands and are easy to use/understand in a hybrid script. For example, the Exchange Online command `get-mailbox` is `get-EOLmailbox` when you use this module to launch it, as shown later.

Note The following native Office 365 modules are necessary for the Office 365 functions in the `vsadmin` module to work, or it will ask you to install them.

- **ExchangeOnlineManagement,** `www.powershellgallery.com/packages/ExchangeOnlineManagement`
- **Sharepoint Online,** `www.microsoft.com/en-ca/download/details.aspx?id=35588`
- **MSOnline Module,** `www.powershellgallery.com/packages/MSOnline`
- **Skype Online (Retired)** All cmdlets are under the Teams Module at `www.powershellgallery.com/packages/MicrosoftTeams`

For example, you can use `LaunchEOL` if you just want to connect to Office 365 Exchange online. It will prompt you for authentication and, once authenticated, you will get connected. It will check if the Exchange Online Management Shell is installed on your computer or not. If not, it will provide you with a hint. See Figure 5-9.

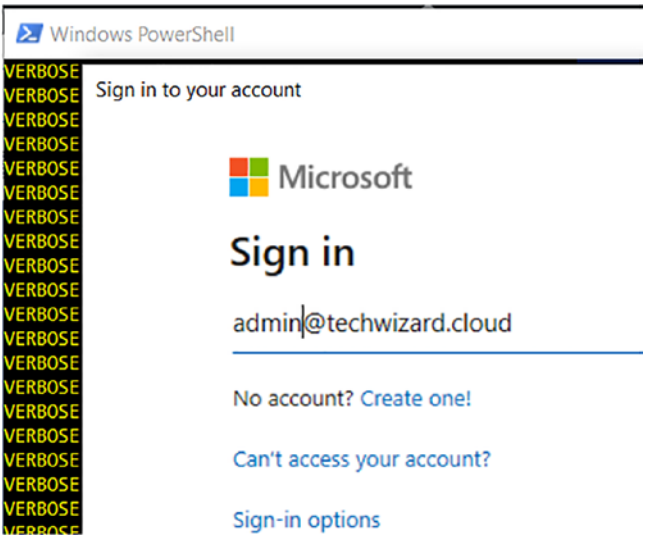


Figure 5-9. *Authentication prompt by Office 365*

Figure 5-10 shows that you are connected and can use the Exchange commands.

```
Windows PowerShell
C:\temp> LaunchEOL
C:\temp> Get-EOLMailbox

Name                        Alias           Database                 ProhibitSendQuota
----                        -----           --------                 -----------------
Amit.Maurya                 Amit.Maurya     NAMPR10DG086-db088       99 GB (106,300,44...
DiscoverySearchMailbox...   DiscoverySea... NAMPR10DG055-db084       50 GB (53,687,091...
DmarcFailures               DmarcFailures   NAMPR10DG194-db067       49.5 GB (53,150,2...
DmarcReports                DmarcReports    NAMPR10DG004-db025       49.5 GB (53,150,2...
GGaba                       GGaba           NAMPR10DG008-db069       99 GB (106,300,44...
info                        info            NAMPR10DG020-db020       49.5 GB (53,150,2...
Jgaba                       Jgaba           NAMPR10DG139-db134       99 GB (106,300,44...
PGaba                       PGaba           NAMPR10DG131-db097       99 GB (106,300,44...
SukhijaPradip               ps4cloud        NAMPR10DG264-db041       99 GB (106,300,44...
rohit.singla                rohit.singla    NAMPR10DG064-db107       99 GB (106,300,44...
SGaba                       SGaba           NAMPR10DG009-db067       99 GB (106,300,44...
TKhetarpal                  TKhetarpal      NAMPR10DG271-db112       99 GB (106,300,44...
sukhijavikas                sukhijavikas    NAMPR10DG194-db051       99 GB (106,300,44...
```

Figure 5-10. *Exchange Online Management Shell prefixed Get-MailBox command*

If you want to use these commands in a script without entering a password every time (a technique you will learn after finishing this book), `LaunchEOL -Credential` can be used by passing PS credentials.

Similarly, you can use other functions because they are designed in a similar manner. For example, use this code and see the results in Figure 5-11:

```
LaunchSPO –orgName techwizard
```

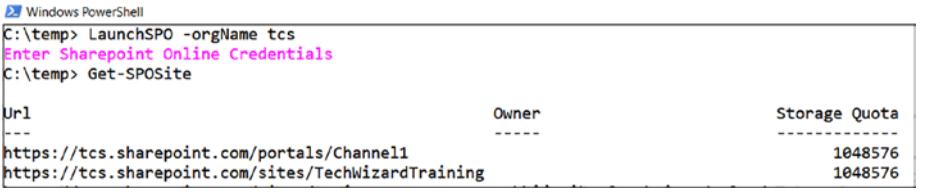


Figure 5-11. *Showing a connection to SharePoint Online using LaunchSPO*

Tip Pressing Tab on a keyboard after pressing the hyphen will show you the parameters available for any function in PowerShell.

To disconnect the session, you can use the following functions:

```
RemoveEOL/RemoveSOL/RemoveSPO, etc.
```

Other good functions that system administrators really like are the `LaunchEXOnprem/RemoveEXOnprem` functions as they are for on-premise Exchange servers. To connect to an Exchange on-premise server from your network, use this code:

```
LaunchEXOnprem -psurl http://exchangeserver.techwizard.cloud/
Powershell
```

or

```
LaunchEXOnprem -ComputerName exchangeserver.techwizard.cloud
```

To disconnect, use the same technique you used for Office 365 functions:

```
RemoveEXOnprem -computername exchangeserver.techwizard.cloud
```

Let's now discuss generic functions inside this module. In Chapter 2, `Write-Log`, `Set-recyclelogs`, `start-progressbar` and other cheat function were shared. These functions are part of this module as well, so you do not have to copy and paste them in your scripts if you are importing this module in the script. See Listing 5-4 and Figure 5-12.

Listing 5-4. Importing vsadmin and Using the Write-Log Function

```
Import-Module vsadmin

$log = Write-Log -Name "log_file" -folder logs -Ext log

Write-Log -Message "Information.........Script" -path
$log  #default  will log as information
Write-Log -Message "warning.........Message" -path $log
-Severity Warning #you can display warning using the severity
Write-Log -Message "error.........Error" -path $log -Severity
error #you can display error using the severity
```

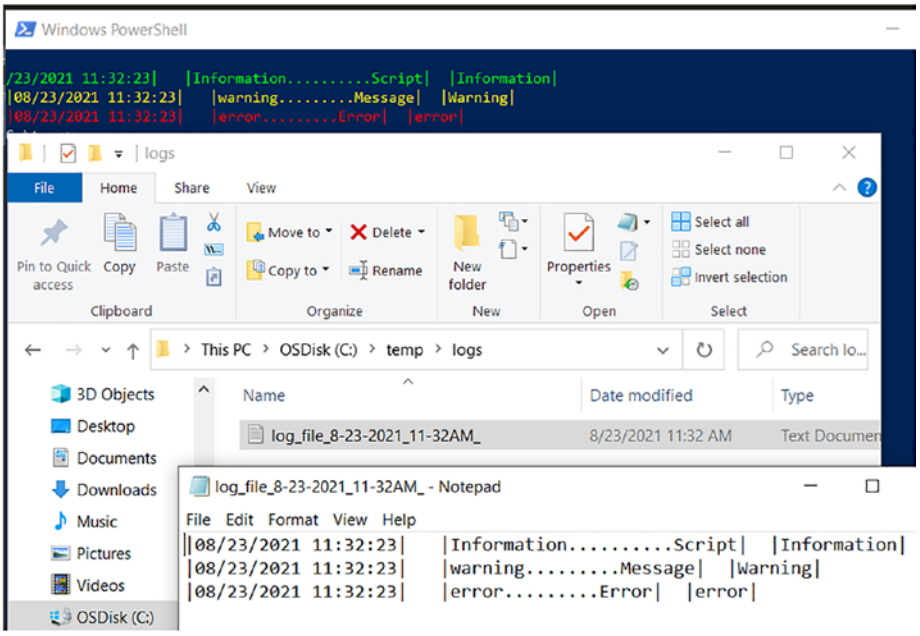


Figure 5-12. *Showing the result of executing Listing 5-4*

In a similar fashion, you can create a CSV file:

```
$report = Write-Log -Name "log_Enable" -folder reports -Ext csv
```

I will not get into the other functions that have been shared in previous chapters. I just wanted to show that they can all be used in this manner as well.

Encrypting a Password (vsadmin)

You can encrypt a password and later utilize in it the script to connect to online services like Office 365 and Azure as follows (see Figure 5-13 for the result):

```
Save-EncryptedPassword -password "testpassword" -path c:\temp\
password1.txt
```

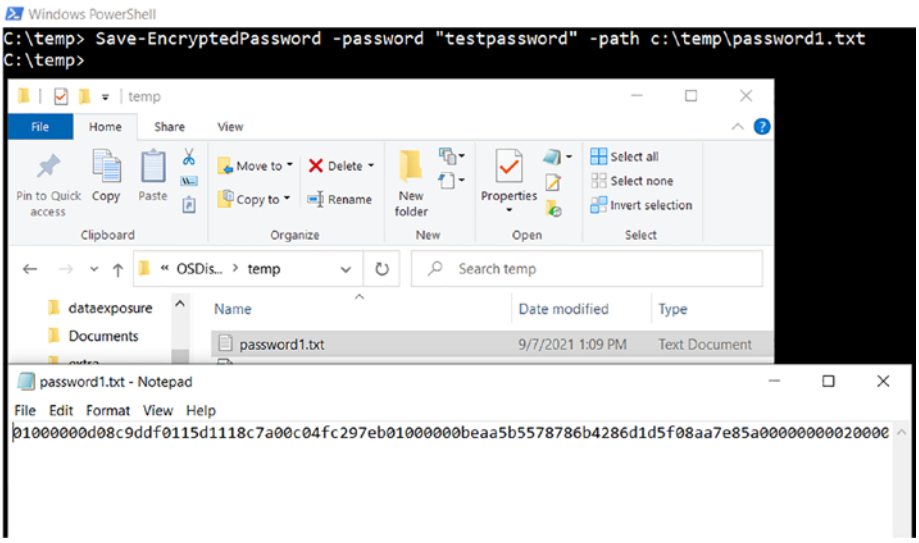


Figure 5-13. *Encrypting a password using the save-encrypted command*

get-auth is another important function that you can use in your scripts to read credentials from the encrypted text file that you created above. You use it as follows:

```
$cred = get-auth -userId sukhija@techwizard.cloud -passwordfile
c:\temp\password1.txt
$pwd = $cred[0] ### credentials that can be used inside csom /
api calls.
$pscredential = $cred[1] ###credentials that can be used for
functions that supports ps credentials.
```

or

```
$cred = get-auth -userId sukhija@techwizard.cloud
-password  "encryptedpassword"
$pwd = $cred[0] ### credentials that can be used inside csom /
api calls.
```

```
$pscredential = $cred[1] ###credentials that can be used for
functions that supports ps credentials.
```

Let's use a small cheat code snippet to connect to Office 365 using the PS credentials saved in the file and export a CSV report on mailboxes. (This can be modified and scheduled as per your needs.) See Listing 5-5 and Figure 5-14.

Listing 5-5. Code Showing Use of PS Credentials

```
Import-Module vsadmin

$cred = get-auth -userId sukhija@techwizard.cloud -passwordfile
"c:\temp\password1.txt"  #getcredentials that you created using
Save-EncryptedPassword
$pscredential = $cred[1] ###credentials that can be used for
functions that supports ps credentials.

LaunchEOL -Credential $pscredential
$data = Get-EOLMailbox -ResultSize unlimited | Select Name,Wind
owsEmailAddress,IssueWarningQuota, ProhibitSendQuota,ProhibitSe
ndReceiveQuota #fetch the required data from exchange online

$data | Export-Csv "c:\temp\mailboxes.csv" -NoTypeInformation
#export the data in csv format

RemoveEOL #disconnect the exchange online session
```

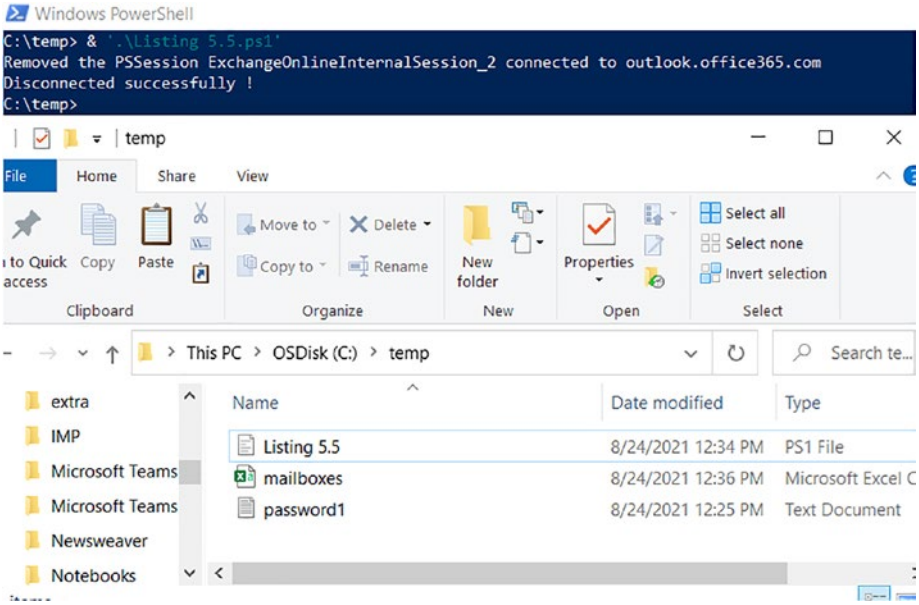


Figure 5-14. *Exporting the mailboxes data in a CSV file*

Use `Get-IniContent` to read ini files and then use the values in scripts. See Figure 5-15.

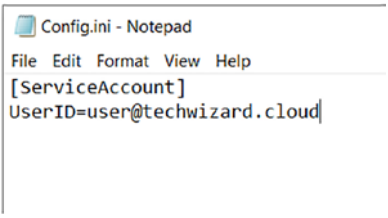


Figure 5-15. *Example INI file*

You can see an example of the following code in Figure 5-16:

```
$inifile = "c:\temp\config.ini"
$readini = Get-IniContent $inifile
$User = $readini["ServiceAccount"].UserID
```

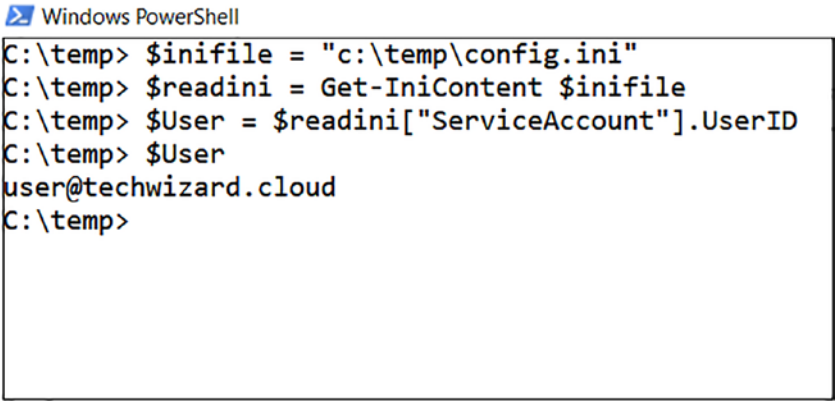


Figure 5-16. *Showing how to read an INI file using Get-IniContent*

Another useful day-to-day function is `Save-CSV2Excel`. As the name suggests, it converts a CSV file to formatted Excel:

```
Save-CSV2Excel -CSVPath C:\data\auditmbx.csv -Exceloutputpath
c:\data\audit.xlsx
```

Random complex passwords are essential in the system administrator world. So why use the Web or a tool to generate them when you can do so with the `New-RandomPassword` function from the `vsadmin` module? See this example in Figure 5-17:

```
New-RandomPassword – NumberofChars 9
```

It has been coded with 5, 9 ,14, and 20.

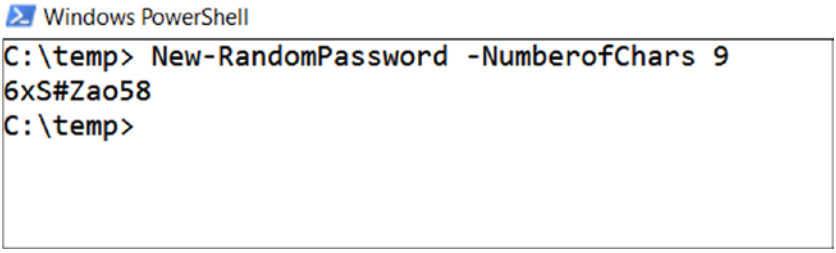


Figure 5-17. *Showing the generation of a random password*

Last but not least, I'll share some Active Directory functions that are not available out of the box from the Active Directory module.

Get-ADGroupMembersRecursive can extract group members recursively, but the AD module is required:

```
Get-ADGroupMembersRecursive -Groups "Test Nested Group"
```

You can read more about this function at https://techwizard.cloud/2020/10/24/get-ad-group-members-recursively/.

Get-ADUserMemberOf can check if a user is a member of a group or not, but the AD module is required:

```
Get-ADUserMemberOf -User "User" -Group "Group"
```

It returns `true` if the user is a member of a group or else it returns `false`. You can read more about this function at https://techwizard.cloud/2020/12/31/check-if-ad-user-is-member-of-group/.

Summary

In this chapter, you learned how to use modules in PowerShell. Modules are like batteries. Without them it is hard to script any product using PowerShell. I also shared a cheat system administration module (`vsadmin`) which has many daily use functions/cmdlets.

CHAPTER 6

Sending Email

Sending email is an important aspect of scripting. Say you want to send alerts if a script results in an error, or you want to send bulk emails without utilizing any bulk email tool.

PowerShell has a simple and effective out-of-the-box cmdlet for this purpose right from PowerShell v2. The following is an example of `Send-MailMessage`, which is prevalent in PowerShell:

```
Send-MailMessage -SmtpServer "smtpserver" -From "DoNotReply@
labtest.com" -To "sukhija@techwizard.cloud " -Subject "Error
exception occured " -Body "body of the message"
```

If you are still using PowerShell 1.0 (which is highly unlikely), you can use the code in Listing 6-1 as it works on all versions of PowerShell.

Listing 6-1. Sending a Message with Powershell v1

```
$smtpserver = "smtp.lab.com"
$to = "sukhija@techwizard.cloud"
$from = "DonotReply@labtest.com"
$file = "c:\file.txt" #for attachment
$subject = "Test Subject"

$message = new-object Net.Mail.MailMessage
$smtp = new-object Net.Mail.SmtpClient($smtpserver)
$message.From = $from
```


V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_6

```
$message.To.Add($to)

$att = new-object Net.Mail.Attachment($file)
$message.IsBodyHtml = $False
$message.Subject = $subject
$message.Attachments.Add($att)
$smtp.Send($message)
```

Formatting a Message Body

There are situations in which you want to send a properly formatted email body instead of just a one-liner email. You can use the cheat code in Listing 6-2 for this purpose. You can see the result in Figure 6-1.

Listing 6-2. Sending a Formatted Message Body

```
$smtpserver = "smtp.lab.com"
$to = "sukhija@techwizard.cloud"
$from = "DonotReply@labtest.com"
$subject = "Test Subject"

$message = @"
Hello,

Line.........................1

Line.........................2

Line.........................3

"@
Send-MailMessage -SmtpServer $smtpserver -From $from -To $to
-Subject $subject -Body $message
```

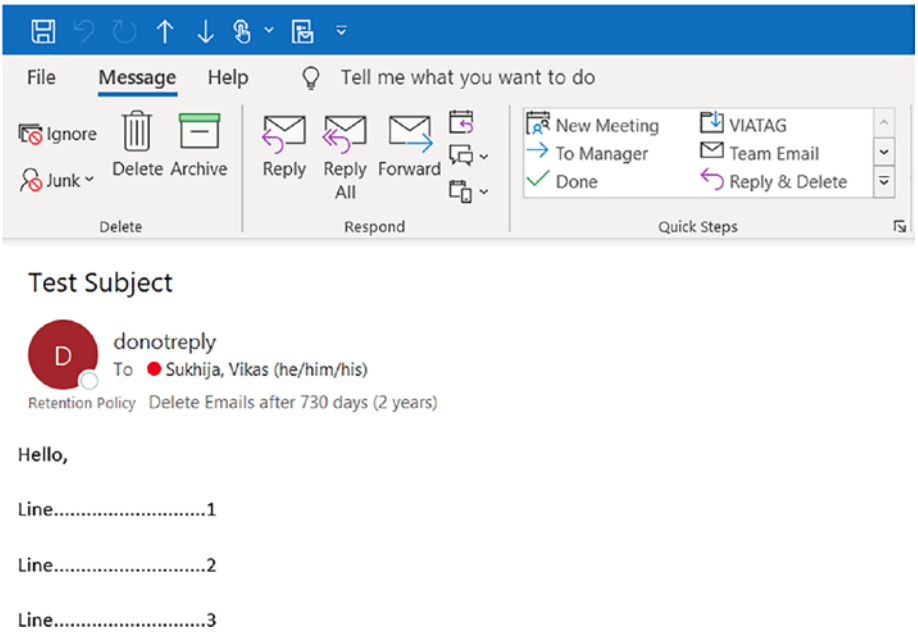


Figure 6-1. *The result of Listing 6-2*

Sending HTML

You can send fancy HTML emails using PowerShell if you use another cheat tip if you are not familiar with HTML. Log on to the HTML Online Editor (shown in Figure 6-2) to create the HTML: `https://html-online.com/editor/`.

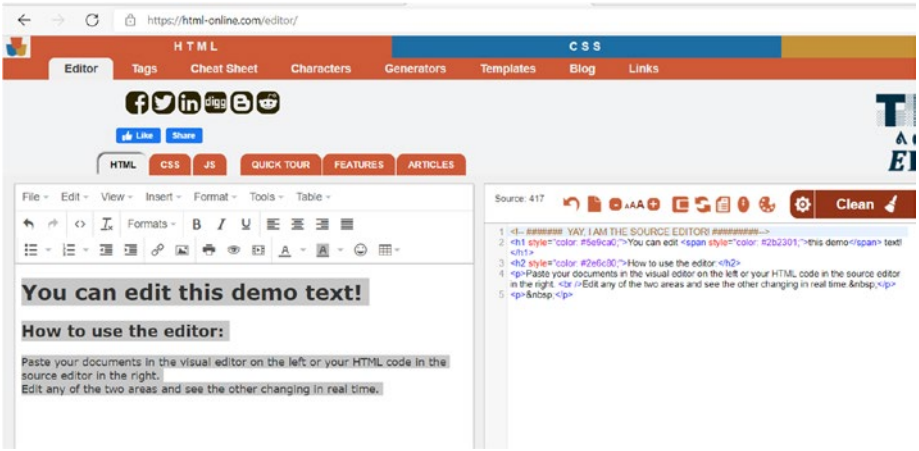


Figure 6-2. *The HTML Online Editor*

Create some HTML content and use the code in Listing 6-3. You can see the result in Figure 6-3.

Listing 6-3. Sending HTML-Formatted Email

```
$smtpserver = "smtp.lab.com"
$to = "sukhija@techwizard.cloud"
$from = "DonotReply@labtest.com"
$subject = "Test Subject"

$message = @"
<!-- ########  YAY, I AM THE SOURCE EDITOR! #########-->
<h1 style="color: #5e9ca0;">You can edit <span style="color:
#2b2301;">this demo</span> text!</h1>
<h2 style="color: #2e6c80;">How to use the editor:</h2>
<p>Paste your documents in the visual editor on the left or
your HTML code in the source editor in the right. <br />Edit
any of the two areas and see the other changing in real
time. </p>
```

```
<p> </p>
"@

Send-MailMessage -SmtpServer $smtpserver -From $from -To $to
-Subject $subject -Body $message -BodyAsHtml
```

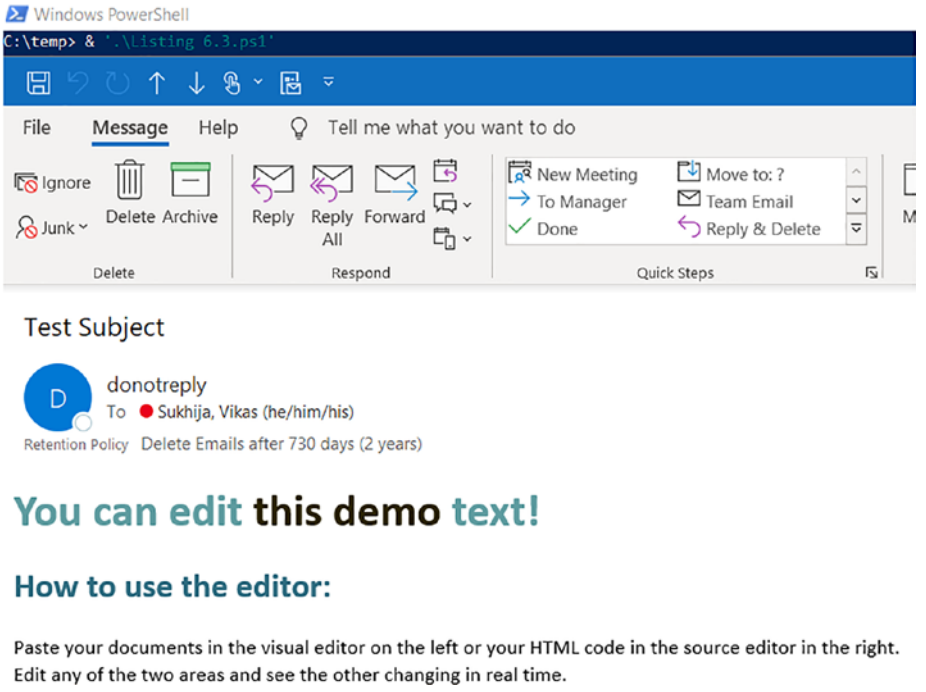


Figure 6-3. *The result of executing Listing 6-3*

Summary

In this chapter, you learned how to send email using PowerShell. You can use this knowledge in the real world to send bulk emails or send email alerts when some task/process/script fails or errors out.

CHAPTER 7

Error Reporting

For successful scripting, error reporting is a must-have. If there is an error, you want to report it to the admin or the owner of the process/automation. PowerShell has different ways to do error reporting. The most common way is the `$error` variable as it contains all the errors that have occurred in the session. Let's look at some cheat code examples that you can utilize. You can log errors or send them via email.

Reporting Errors Through Email

Below is the code that can be used to send errors via email. You can insert this code into your script so that if the script results in an error, it is sent via email.

`$error` is the default variable in PowerShell that contains the error if it occurs during the execution of the code. In Listing 7-1, you check if `$error` is not equal to null, and then send the error in an email.

After sending the error email, if you want to clear the error, you can use `$error.clear()` so that if you are using iterations, the error is not sent again if it does not occur in the next iteration.

Listing 7-1. Sending Errors via Email

```
$from = "donotreply@lab.com"
$to="vikas@lab.com"
$subject = "Error has occured"
```


V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_7

```
$smtpServer="smtp.lab.com"

if ($error)
      {
Send-MailMessage -SmtpServer $smtpserver -From $from -To $to
-Subject $subject -Body $error[0].ToString()
$error.clear()
       }
```

Listing 7-1 is missing one important thing. It just sends the last error, but what if you want to send the full error, which is actually an array and `send-mailmessage` is not able to send it effectively? You can utilize the `Send-Email` function in Listing 7-2, which is also part of the `vsadmin` module that was shared in the modules chapter.

Listing 7-2. Send-Email Function to Send an $error Array

```
function Send-Email
{
  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $true)]
    $From,
    [Parameter(Mandatory = $true)]
    [array]$To,
    [array]$bcc,
    [array]$cc,
    $body,
    $subject,
    $attachment,
    [Parameter(Mandatory = $true)]
    $smtpserver
  )
```

```
$message = New-Object System.Net.Mail.MailMessage
$message.From = $From
if ($To -ne $null)
{
  $To | ForEach-Object{
    $to1 = $_
    $to1
    $message.To.Add($to1)
  }
}
if ($cc -ne $null)
{
  $cc | ForEach-Object{
    $cc1 = $_
    $cc1
    $message.CC.Add($cc1)
  }
}
if ($bcc -ne $null)
{
  $bcc | ForEach-Object{
    $bcc1 = $_
    $bcc1
    $message.bcc.Add($bcc1)
  }
}
$message.IsBodyHtml = $true
if ($subject -ne $null)
{$message.Subject = $subject}
if ($attachment -ne $null)
```

```
  {
    $attach = New-Object Net.Mail.Attachment($attachment)
    $message.Attachments.Add($attach)
  }
  if ($body -ne $null)
  {$message.body = $body}
  $smtp = New-Object Net.Mail.SmtpClient($smtpserver)
  $smtp.Send($message)
}
```

Once you have imported the function from Listing 7-2, you can utilize the code in Listing 7-3, which is similar to the code in Listing 7-1. The only change is utilizing `Send-Email` instead of the built-in `Send-MailMessage`.

Listing 7-3. Sending $error Array in an Email

```
$from = "donotreply@lab.com"
$to="vikas@lab.com"
$subject = "Error has occured"
$smtpServer="smtp.lab.com"

if ($error)
      {
  Send-Email -smtpserver $smtpServer -From $from -To $to
  -subject $subject -body $error
  $error.clear()
       }
```

Logging Everything Including Errors

There is an built-in PowerShell cmdlet that you can use at the beginning of your script and stop at the end of the script. This is often called transcript logging.

```
Start-transcript  # at the beginning of the script
Stop-transcript  # at the end of the script
```

This log will by default get stored in the running account's My Documents folder. To store the transcript at a different location, you can specify the path parameter as shown in Figure 7-1.

```
$log = "c:\data\log.txt"
Start-transcript  -path $log # at the beginning of the script
Stop-transcript  # at the end of the script
```

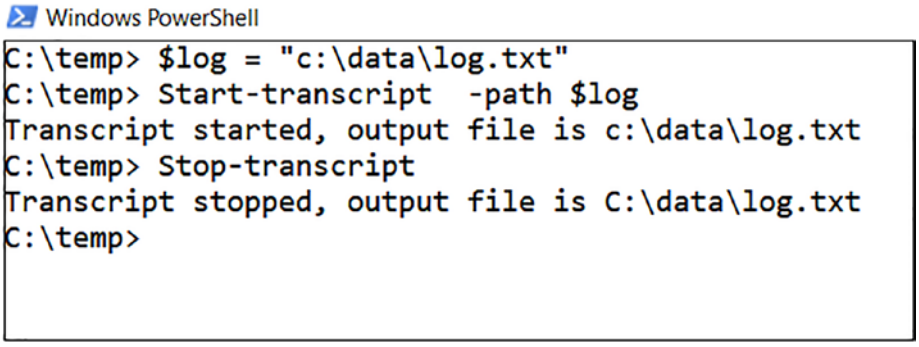


Figure 7-1. *Showing the transcript log in PowerShell*

Logging Errors to a Text File

You can also just log errors in text files. This cheat guide describes a `Write-Log` function; the same can be utilized to log errors in text files. The result is shown in Figure 7-2.

```
$log = Write-Log -Name "Errorlog" -folder "logs" -Ext "log"
write-log -message "error is $error" -path $log -Severity error
```

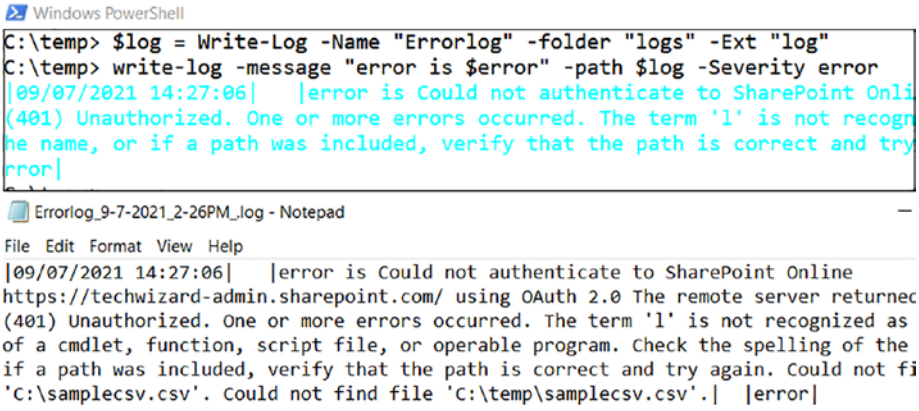


Figure 7-2. *Showing the error logging in a text file*

Summary

In this chapter, you learned how to report errors when they occur, how to write them in log files, how to send them through email, and how to capture the whole PowerShell session with the `start-transcript` cmdlet.

CHAPTER 8

Reporting

Reports are an important aspect of day-to-day system administration. Sometimes we present reports to our managers and other times we utilize them to evaluate and improve our own work.

Reports can be in the form of CSV, HTML, Excel, and more. The most common form is a CSV report because it's universal and can be converted to other forms like Excel easily from the application itself.

CSV Report

`Export-CSV` is the built-in PowerShell cmdlet that can be used to export the data to a CSV file. You can simply **pipe select and then export** as shown in the following code that shows the export of certain attributes of users' mailboxes from Exchange Server (to be run in the Exchange shell):

```
Get-Mailbox -ResultSize unlimited | Select Name,identity, Windo
wsEmailAddress,Database,ProhibitSendQuota,ProhibitSendReceiveQu
ota,IssueWarningQuota | export-csv c:\mailboxes.csv -notypeinfo
```

There are many complex situations where you need scripting code to format the data in the right manner. Listing 8-1 and Figure 8-1 show where you have a list of users in a text file and you want to export their Active Directory attributes and some Exchange attributes in a CSV file. In this case, get, select, and export are not possible. This code can be used in all such situations where you want to export the data in the CSV form but a simple operation is not possible.

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_8

You want to export attributes `Name,identity,WindowsEmailAddress, Database,ProhibitSendQuota,ProhibitSendReceiveQuota,` and `IssueWarningQuota` and also attributes `employeeid, l, C` from Active Directory.

Note The Exchange and AD modules are both required. You need to connect to them. The script will fail if they are not loaded.

To load them, use your knowledge from previous chapters.

Load the Exchange on-premise shell using `vsadmin launchexonprem`:

```
LaunchEXOnprem -ComputerName ExchangeServer
```

For Active Directory, you can use

```
Import-Module Activedirectory
```

Listing 8-1. Exporting to CSV When Fetching From Multiple Sources

```
$collection=@() #array to collect report data
$data = get-content .\users.txt #read samaccountname from text file
$data | foreach-object{
  $coll = "" | Select Name,identity,WindowsEmailAddress,Database,
  ProhibitSendQuota,ProhibitSendReceiveQuota,IssueWarningQuota,
  employeeid, l,C  #values needed in report

  $getmbx = get-mailbox -identity $_
  $getaduser = get-aduser -identity $_ -properties employeeid, l,C
  $coll.Name = $getmbx.Name
  $coll.identity = $getmbx.identity
  $coll.WindowsEmailAddress = $getmbx.WindowsEmailAddress
  $coll.Database= $getmbx.Database
  $coll.ProhibitSendQuota = $getmbx.ProhibitSendQuota
  $coll.ProhibitSendReceiveQuota = $getmbx.ProhibitSendReceiveQuota
```

```
  $coll.IssueWarningQuota = $getmbx.IssueWarningQuota
  $coll.employeeid = $getaduser.employeeid #note difference here
  $coll.l = $getaduser.l
  $coll.c = $getaduser.c
  $collection+=$coll #add the collected values to the collecttion
  array
}
#now export to CSV file
$collection | Export-Csv .\report.csv -NoTypeInformation
```

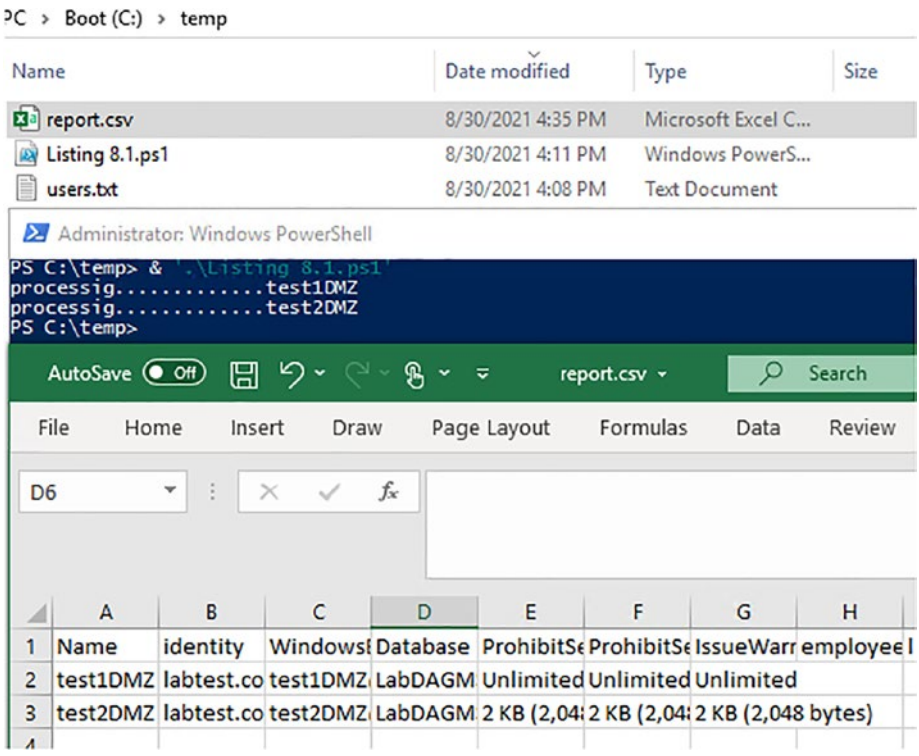


Figure 8-1. *Showing the execution result of Listing 8-1*

Another important aspect of CSV reporting is to export multi-valued attributes. Here is an example of extracting recipients (which is a multi-valued attribute) in Exchange tracking logs:

```
@{Name="Recipents";Expression={$_.recipients}}
```

See Listing 8-2 for an example of extracting recipient values from Exchange transport logs.

Listing 8-2. Example Code Showing How to Export Multi-Value Attributes

```
Get-transportserver | Get-MessageTrackingLog -Start"03/09/2015
00:00:00 AM" -End"03/09/2015 11:59:59 PM" -sender "vikas@lab.com"
-resultsize unlimited | `
select-object Timestamp,clientip,ClientHostname,ServerIp,
ServerHostname,sender,EventId,MessageSubject, TotalBytes ,
SourceContext,ConnectorId,Source, `
InternalMessageId , MessageId ,@{Name="Recipents";Expression=
{$_.recipients}} | `
export-csv c:\track.csv
```

Excel Reporting

Although CSV reports are fine for most purposes, there are situations in which you want to share the data with your managers so converting the CSV file to Excel is a much-needed script. I will share two methods for doing the same.

The first method exists in the `vsadmin` module that was shared in the modules chapter.

Note Excel should be installed on the machine to use this method.

Listing 8-3 shows the code of the Save-CSV2Excel function in case you do not have the vsadmin module installed or do not want to use it.

Listing 8-3. Cheat Code for the Save-CSV2Excel Function

```
Function Save-CSV2Excel
{
  [CmdletBinding()]
  Param(
    [Parameter(Mandatory = $true,Position = 1)]
    [ValidateScript({
          if(-Not ($_ | Test-Path) ){throw "File or folder does
          not exist"}
          if(-Not ($_ | Test-Path -PathType Leaf) ){throw "The
          Path argument must be a file. Folder paths are not
          allowed."}
          if($_ -notmatch "(\.csv)"){throw "The file specified
          in the path argument must be either of type csv"}
          return $true
    })]
    [System.IO.FileInfo]$CSVPath,

    [Parameter(Mandatory = $true)]
    [ValidateScript({
          if($_ -notmatch "(\.xlsx)"){throw "The file specified
          in the path argument must be either of type xlsx"}
          return $true
    })]
    [System.IO.FileInfo]$Exceloutputpath
  )
  ####### Borrowed function from Lloyd Watkinson from script
  gallery##
  Function Convert-NumberToA1
```

```
  {
    Param([parameter(Mandatory = $true)]
    [int]$number)

    $a1Value = $null
    While ($number -gt 0)
    {
      $multiplier = [int][system.math]::Floor(($number / 26))
      $charNumber = $number - ($multiplier * 26)
      If ($charNumber -eq 0) { $multiplier-- ; $charNumber = 26
}
      $a1Value = [char]($charNumber + 64) + $a1Value
      $number = $multiplier
    }
    Return $a1Value
  }
  ##################Start converting
  excel#########################
  $importcsv = Import-Csv $CSVPath
  $countcolumns = ($importcsv |
    Get-Member |
  Where-Object{$_.membertype -eq "Noteproperty"}).count
  ##################call Excel com object ##############
  $xl = New-Object -comobject excel.application
  $xl.visible = $false
  $Workbook = $xl.workbooks.open($CSVPath)
  $Workbook.SaveAs($Exceloutputpath, 51)
  $Workbook.Saved = $true
  $xl.Quit()
  #############Now format the Excel###################
  timeout.exe 10 #wait for 10 seconds before saving
  $xl = New-Object -comobject excel.application
```

```
  $xl.visible = $false
  $Workbook = $xl.workbooks.open($Exceloutputpath)
  $worksheet1 = $Workbook.worksheets.Item(1)
  for ($c = 1; $c -le $countcolumns; $c++) {$worksheet1.Cells.
  Item(1, $c).Interior.ColorIndex = 39}
  $colvalue = (Convert-NumberToA1 $countcolumns) + "1"
  $headerRange = $worksheet1.Range("a1", $colvalue)
  $null = $headerRange.AutoFilter()
  $null = $headerRange.entirecolumn.AutoFit()
  $worksheet1.rows.item(1).Font.Bold = $true
  $Workbook.Save()
  $Workbook.Close()
  $xl.Quit()
  $Null = [System.Runtime.Interopservices.Marshal]::Release
  ComObject($xl)
  ##############################################################
  #############
}#Write-CSV2Excel
```

Let's use the CSV report from Listing 8-1 and convert it to Excel using `Save-CSV2Excel`. See Figure 8-2.

```
Save-CSV2Excel -CSVPath c:\temp\report.csv -Exceloutputpath
c:\temp\report.xlsx
```

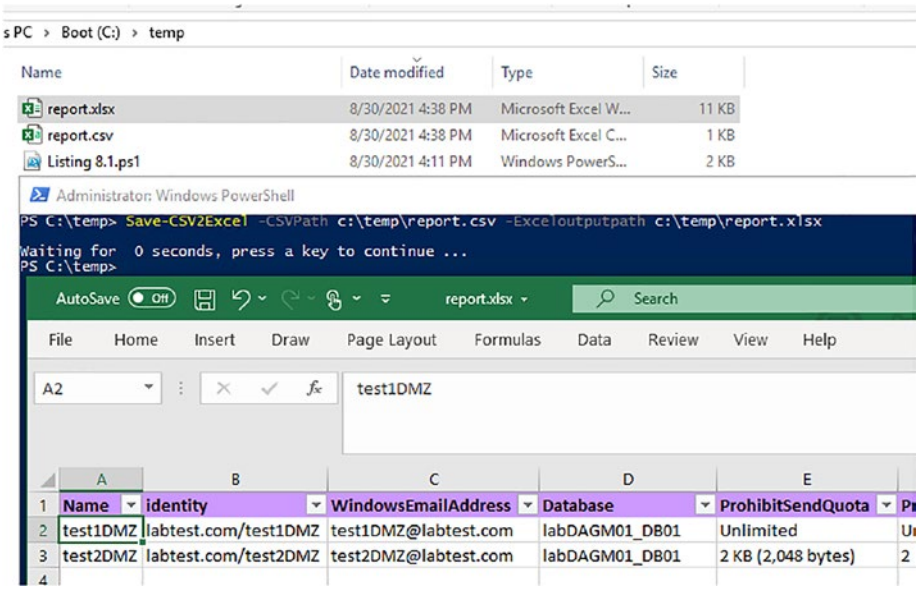


Figure 8-2. *Showing a CSV-to-Excel conversion*

There is a module named ImportExcel. It is one of the most popular modules in the PowerShell Gallery. You can utilize this module to directly convert variables to Excel. Get it from www.powershellgallery.com/packages/ImportExcel. Install the module on your machine and then import it to utilize it:

```
Install-Module -Name ImportExcel
```

Let's use the same report and use this new module to convert it into Excel. The advantage of using this module is that it does not require Excel to be installed on the machine. See Figure 8-3.

```
Import-Module -Name ImportExcel

$data = Import-Csv .\report.csv
$data | Export-Excel -Path c:\temp\report.xlsx
```

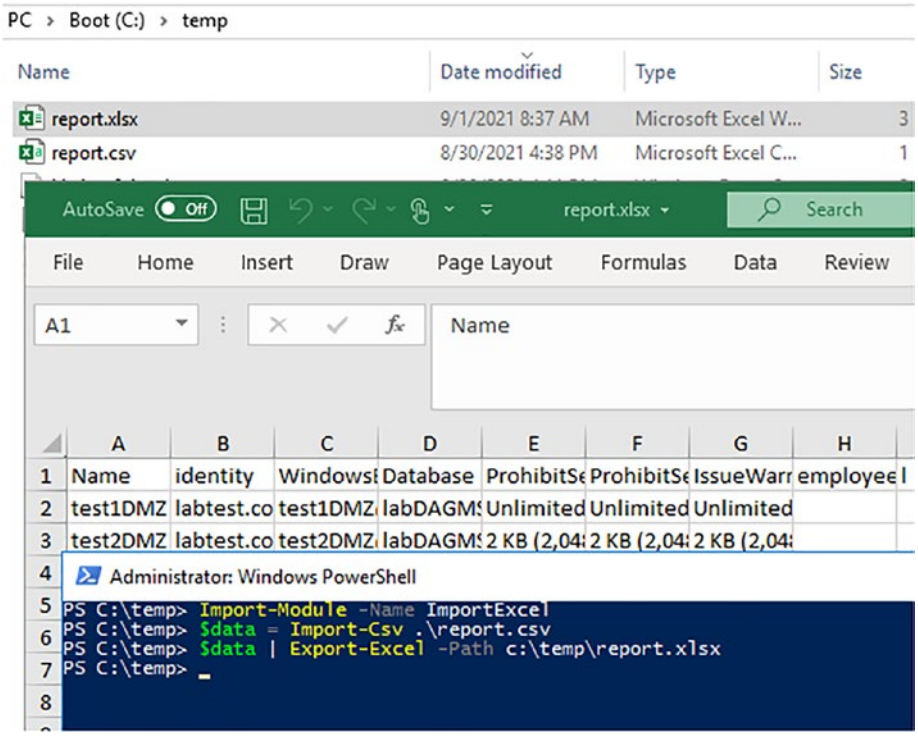
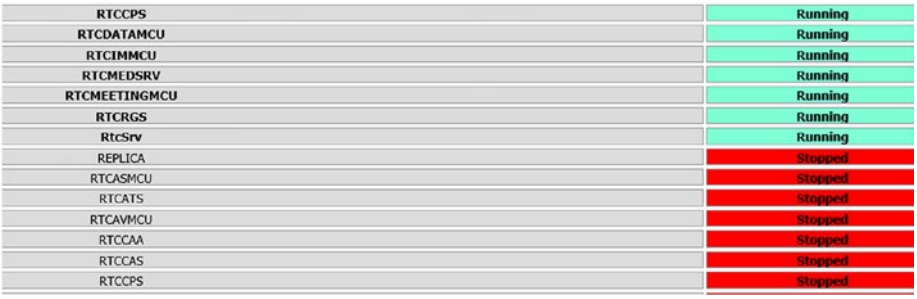


Figure 8-3. *Using the Import-Excel module*

There are lot of other parameters inside that function like the formatting of Excel, which I leave to you to explore!

HTML Reporting

It would be wonderful if we could create HTML dashboards with PowerShell ☺ that can show traffic light-type signals. For example, if a service is down, it shows red. Otherwise, it shows green. See Figure 8-4.



RTCCPS	Running
RTCDATAMCU	Running
RTCIMMCU	Running
RTCMEDSRV	Running
RTCMEETINGMCU	Running
RTCRGS	Running
RtcSrv	Running
REPLICA	Stopped
RTCASMCU	Stopped
RTCATS	Stopped
RTCAVMCU	Stopped
RTCCAA	Stopped
RTCCAS	Stopped
RTCCPS	Stopped

Figure 8-4. *An HTML table report*

Listing 8-4 is a template for HTML coding that you can use inside scripts and do traffic light-type operations based on conditions.

Listing 8-4. Template for HTML Coding

```
$report = $reportpath
Clear-Content $report
##################HTml Report
Content#############################
Add-Content $report "<html>"
Add-Content $report "<head>"
Add-Content $report "<meta http-equiv='Content-Type'
content='text/html; charset=iso-8859-1'>"
Add-Content $report '<title>Exchange Status Report</title>'
add-content $report '<STYLE TYPE="text/css">'
add-content $report  "<!--"
add-content $report  "td {"
add-content $report  "font-family: Tahoma;"
add-content $report  "font-size: 11px;"
add-content $report  "border-top: 1px solid #999999;"
add-content $report  "border-right: 1px solid #999999;"
add-content $report  "border-bottom: 1px solid #999999;"
add-content $report  "border-left: 1px solid #999999;"
```

```
add-content $report  "padding-top: 0px;"
add-content $report  "padding-right: 0px;"
add-content $report  "padding-bottom: 0px;"
add-content $report  "padding-left: 0px;"
add-content $report  "}"
add-content $report  "body {"
add-content $report  "margin-left: 5px;"
add-content $report  "margin-top: 5px;"
add-content $report  "margin-right: 0px;"
add-content $report  "margin-bottom: 10px;"
add-content $report  ""
add-content $report  "table {"
add-content $report  "border: thin solid #000000;"
add-content $report  "}"
add-content $report  "-->"
add-content $report  "</style>"
Add-Content $report "</head>"
Add-Content $report "<body>"
add-content $report  "<table width='100%'>"
add-content $report  "<tr bgcolor='Lavender'>"
add-content $report  "<td colspan='7' height='25'
align='center'>"
add-content $report  "<font face='tahoma' color='#003399'
size='4'><strong>DAG Active Manager</strong></font>"
add-content $report  "</td>"
add-content $report  "</tr>"
add-content $report  "</table>"
add-content $report  "<table width='100%'>"
Add-Content $report  "<tr bgcolor='IndianRed'>"
Add-Content $report  "<td width='10%'
align='center'><B>Identity</B></td>"
```

```
Add-Content $report  "<td width='5%' align='center'><B>PrimaryA
ctiveManager</B></td>"
Add-Content $report  "<td width='20%' align='center'><B>Operati
onalMachines</B></td>"
Add-Content $report "</tr>"
###############################Report
Template###################################
add-content $report  "<tr bgcolor='Lavender'>"
add-content $report  "<td colspan='7' height='25'
align='center'>"
add-content $report  "<font face='tahoma' color='#003399'
size='4'><strong>DAG Database Backup Status</strong></font>"
add-content $report  "</td>"
add-content $report  "</tr>"
add-content $report  "</tr>"
add-content $report  "</table>"
add-content $report  "<table width='100%'>"
Add-Content $report "<tr bgcolor='IndianRed'>"
Add-Content $report  "<td width='10%'
align='center'><B>Database</B></td>"
Add-Content $report  "<td width='5%' align='center'><B>BackupIn
Progress</B></td>"
Add-Content $report  "<td width='10%' align='center'><B>Snapsho
tLastFullBackup</B></td>"
Add-Content $report  "<td width='5%' align='center'><B>Snapshot
LastCopyBackup</B></td>"
Add-Content $report  "<td width='10%' align='center'><B>LastFul
lBackup</B></td>"
Add-Content $report  "<td width='5%' align='center'><B>RetainDe
letedItemsUntilBackup</B></td>"
```

```
$dbst= Get-MailboxDatabase | where{$_.MasterType -like
"DatabaseAvailabilityGroup"}

$dbst | foreach{$st=Get-MailboxDatabase $_ -status
  $dbname =  $st.Name
  $dbbkprg = $st.BackupInProgress
  $dbsnpl = $st.SnapshotLastFullBackup
  $dbsnplc= $st.SnapshotLastCopyBackup
  $dblfb = $st.LastFullBackup
  $dbrd = $st.RetainDeletedItemsUntilBackup
    Add-Content $report "<tr>"
    Add-Content $report "<td bgcolor= 'GainsBoro'
    align=center>  <B>$dbname</B></td>"
          Add-Content $report "<td bgcolor= 'GainsBoro'
          align=center>  <B>$dbbkprg</B></td>"
    Add-Content $report "<td bgcolor= 'GainsBoro'
    align=center>  <B>$dbsnpl</B></td>"
          Add-Content $report "<td bgcolor= 'GainsBoro'
          align=center>  <B>$dbsnplc</B></td>"
  if($dblfb -lt $hrs)
  {
    Add-Content $report "<td bgcolor= 'Red'
    align=center>  <B>$dblfb</B></td>"
  }
  else
  {
      Add-Content $report "<td bgcolor= 'Aquamarine'
      align=center>  <B>$dblfb</B></td>"
  }
    Add-Content $report "<td bgcolor= 'GainsBoro'
    align=center>  <B>$dbrd</B></td>"
```

```
    Add-Content $report "</tr>"
}
###############################################################
Add-content $report  "</table>"
Add-Content $report "</body>"
Add-Content $report "</html>"
```

See examples at the following links where this template has been successfully utilized for the Exchange Health Check, AD Health Check, and Monitor Remote services:

https://techwizard.cloud/exchange-2010-health-check/
https://techwizard.cloud/adhealthcheck/
https://techwizard.cloud/monitor-windows-services-status-remotely/

As mentioned, you can use the HTML Online Editor to create HTML and use it in your PowerShell scripts (https://html-online.com/editor/).

Summary

In this chapter, you learned about reporting. CSV, HTML, and Excel are three common report types used by almost all systems. By learning how to run these reports you can easily impress managers with the data they need in a format that is understandable.

CHAPTER 9

Miscellaneous Keywords

In this chapter, you will be introduced to keywords in PowerShell. These keywords can perform data manipulation, which is a crucial part of any scripting or automation operation.

In the next chapter, you will use the knowledge you have gathered in this book to create some practical production scripts using the cheat codes shared in this book.

Split

The `split` keyword can be used to extract data out of a string. Say there is an email address inside a string and you want to get it, or you want to extract some useful information out of a text string.

Let's go through an example to understand it more. Let's extract the first name and last name from an email address string. You will use `split` because it can split the string on any character and convert it into array. You will split the email address on the dot (.). After that, in element 0 of the array you get the first name, but for last name you must split again at character @.

```
$email = "Vikas.Sukhija@labtest.com"
$emsplit = $email.split(".")
```


V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_9

```
$firstname = $emsplit[0]
$lastname = ($emsplit[1] -split "@")
$lastn = $lastname[0]
$emsplit[0] and $lastname[0]
```

See the step-by-step split operation in Figure 9-1 for a better understanding of how to use it.

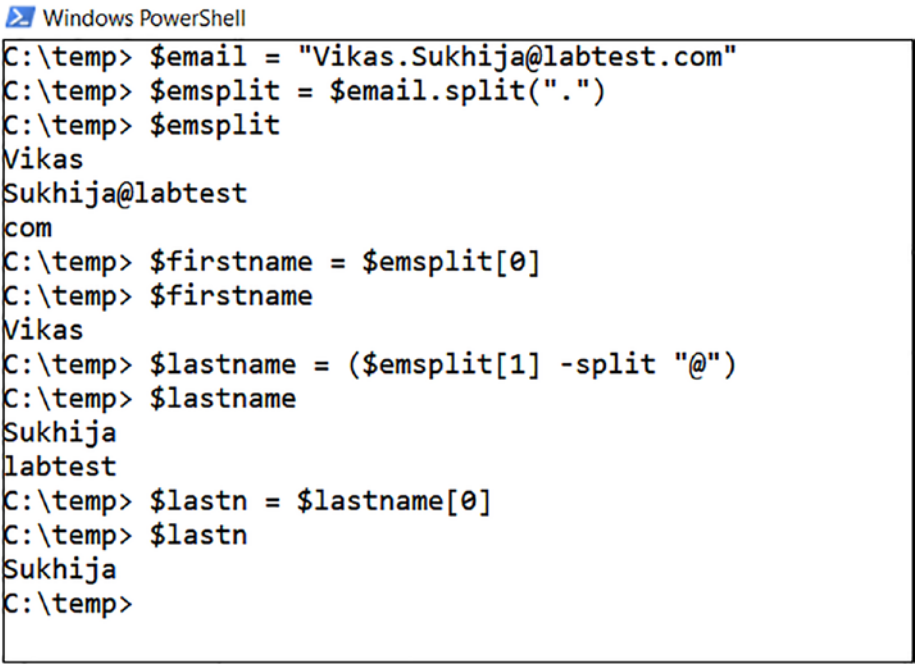


Figure 9-1. *Showing the split operation*

Replace

Another keyword is `replace`. Instead of splitting the string, you can replace the content of the string with other content.

You can use `replace` when you want to replace data in a string. Say you want to add an underscore instead of a dot because you want to update a secondary address. You can use this code and see the result in Figure 9-2:

```
$email = "Vikas.Sukhija@labtest.com"
$emreplace = $email.replace(".","_")
```

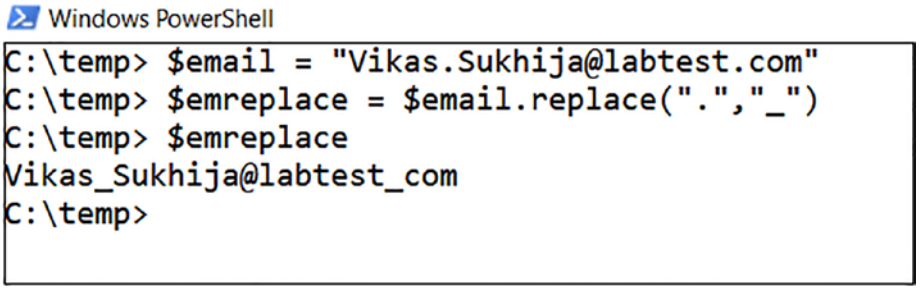


Figure 9-2. *Showing a replace operation*

Select-String

`Select-String` can do wonders because you can use it to find strings inside files. Here is a practical use for it, which I have used many times (while others have struggled and spent ample hours trying to solve): finding the right date and time of an operation from a large number of log files.

Say you have a large number of files inside the `logs` folder and you just want to find the files where the string `error` is present:

```
Get-ChildItem c:\data\logs | Select-String -Pattern "Error"
```

This simple one-liner will search for the string `error` in all of the log files inside the `logs` folder. Figure 9-3 show how it extracted the file name of the file that has the error.

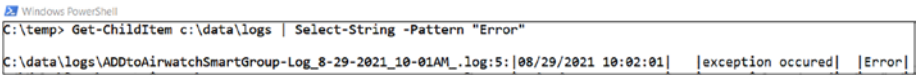


Figure 9-3. *Showing a Select-String operation*

Compare-Object

`Compare-Object` (alias `Compare`) is used many times to compare two files or two arrays. It is faster than comparing the arrays or files using `for` loops. I use it many times for fetching members from a group and comparing them with a text file that has user IDs. This approach fetches only members that are not already part of the group and adds them, instead of processing all members.

Listing 9-1 adds members from one group to another.

Note The Active Directory module is required for this to work.

Listing 9-1. Cheat Code for Adding Members Using Compare-Object

```
##########################fetching group1 ##################
 $collgroup1 = Get-ADGroup -id "group1" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
##########################fetching group2 ##################
 $collgroup2 = Get-ADGroup -id "group2" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
####################compare two groups#####################
```

```
 $change = Compare-Object -ReferenceObject $collgroup1
-DifferenceObject $collgroup2
 $Addition = $change |
 Where-Object -FilterScript {$_.SideIndicator -eq "<="} |
 Select-Object -ExpandProperty InputObject
#######adding only members that are missing in
group2##############
 $Addition | ForEach-Object{
      $sam = $_
      Add-ADGroupMember -identity "group2" -Members $sam
 }
#####################################################################
```

Similarly, you can do a remove operation by using `Compare-Object`, as shown in Listing 9-2.

Listing 9-2. Cheat Code for Removing Members Using Compare-Object

```
############################fetching group1 ##################
 $collgroup1 = Get-ADGroup -id "group1" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
############################fetching group2 ##################
 $collgroup2 = Get-ADGroup -id "group2" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
####################compare two groups###################
 $change = Compare-Object -ReferenceObject $collgroup1
-DifferenceObject $collgroup2
 $Removal = $change |
```

```
 Where-Object -FilterScript {$_.SideIndicator -eq "=>"} |
 Select-Object -ExpandProperty InputObject
####Removing members that are in group2 but not in
group1########
 $Removal | ForEach-Object{
      $sam = $_
      Remove-ADGroupMember -identity "group2" -Members $sam
      -confirm:$false
 }
#####################################################################
```

You can combine both operations in one script and synchronize two groups based on group1 as the anchor. Listing 9-3 shows this operation.

Listing 9-3. Cheat Code for Synchronizing Two Groups Using Compare-Object (Based on group1 as the Anchor)

```
############################fetching group1 ##################
 $collgroup1 = Get-ADGroup -id "group1" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
############################fetching group2 ##################
 $collgroup2 = Get-ADGroup -id "group2" -Properties member |
  Select-Object -ExpandProperty member |
  Get-ADUser |
  Select-Object -ExpandProperty samaccountname
####################compare two groups####################
 $change = Compare-Object -ReferenceObject $collgroup1
-DifferenceObject $collgroup2
$Addition = $change |
 Where-Object -FilterScript {$_.SideIndicator -eq "<="} |
 Select-Object -ExpandProperty InputObject
```

```
 $Removal = $change |
 Where-Object -FilterScript {$_.SideIndicator -eq "=>"} |
 Select-Object -ExpandProperty InputObject
#######adding only members that are missing in group2########
 $Addition | ForEach-Object{
      $sam = $_
      Add-ADGroupMember -identity "group2" -Members $sam
 }

 ####Removing members that are in group2 but not in
group1########
 $Removal | ForEach-Object{
      $sam = $_
      Remove-ADGroupMember -identity "group2" -Members $sam
      -confirm:$false
 }
#################################################################
```

You can also use the other approach, so instead of removing from group2 you just use `ADD-Groupmember` for group1 so you can truly synchronize both groups. Any user object that is not present in group2 but is in group1 should be added to group2, and any user object not present in group1 but in group2 should be added to group1:

```
ADD-ADGroupMember -identity "group1" -Members $sam
```

instead of

```
Remove-ADGroupMember -identity "group2" -Members $sam
-confirm:$false
```

There are other nice tricks you can perform with `Compare-Object`. Say you have two CSV files. One just has email addresses of users; the other has email addresses and other properties. You want all details from CSV file 2 for the users in CSV file one.

Listing 9-4 shows an example for OneDrive properties. There are two CSV files. One contains user email addresses and the other contains email addresses and other properties in other columns.

Listing 9-4. Cheat Code for Merging Two CSV Files Using Compare-Object

```
$importallonedrivesites = import-csv "c:\importonedrives.csv" #
onedrive file with other attributes
$importspofile = import-csv "c:\users.csv" #users email
addresses

$change = Compare-Object -ReferenceObject
$importallonedrivesites   -DifferenceObject $importspofile
-Property owner -IncludeEqual -PassThru  #owner is the column
name for users email addreses
$change | where{$_.SideIndicator -eq "==" -or $_.SideIndicator
-eq "=>"} |
select Owner, Title, url, StorageUsageCurrent, StorageQuota,
StorageQuotaWarningLevel |
Export-Csv "c:\newfile.csv" -NoTypeInformation
```

Summary

In this chapter, you learned about important keywords in PowerShell, which will help you with data manipulation or transformation. This means you can automate information when the data input is in a different format than expected.

CHAPTER 10

Gluing It All Together

This is the last chapter of this book, and in it I will show you how to use the knowledge you have gathered thus far to create a practical script. I will also share some cheat codes from different products that you can utilize for your daily needs.

Here is the scenario: You get a text file from your HR system that contains a list of account names (Figure 10-1) that you want to add to an Active Directory group so that they can access a particular file share where this AD group has permissions, or get some app pushed to their devices based on their membership of that AD group.

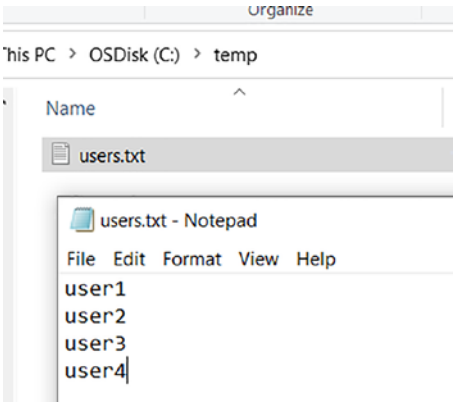


Figure 10-1. *Showing the example users text file*

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1_10

Step 1: Add headers to the script (see Figure 10-2):

```
<#
    .NOTES
    ===========================================================
    Created with:     ISE
    Created on:       9/6/2021 1:46 PM
    Created by:       Vikas Sukhija
    Organization:
    Filename:         ADDUserstoGroupfromText.ps1
    ===========================================================
    .DESCRIPTION
    This will add the users from text file to AD group
#>
```

```
<#
    .NOTES
    ===========================================================
    Created with:       ISE
    Created on:         9/6/2021 1:46 PM
    Created by:         Vikas Sukhija
    Organization:
    Filename:           ADDUserstoGroupfromText.ps1
    ===========================================================
    .DESCRIPTION
    This will add the users from text file to AD group
#>
```

Figure 10-2. *Showing headers in ISE*

Step 2: Import all modules that you will utilize for this script:

1. The `vsadmin` module will make your life easy for the scripting operations.
2. Active Directory modules

If you do not want to use the `vsadmin` module, then just use the functions instead.

```
import-module vsadmin
import-module Activedirectory
```

Step 3: Add some variables and logs for your script:

```
$log = Write-Log -Name "ADDUser2Group-Log" -folder "logs" -Ext "log"
$users = get-content "c:\temp\users.txt"
$Adgroup = "ADgroup1"
$logrecyclelimit = "60" #to recycle the logs after 60 days
```

Step 4: Start the actual operation:

```
Write-Log -Message "Start..............script" -path $log
$users | foreach-object{
  $user = $_.trim() #triming fro any whitespace
  Write-Log -Message "Processing..........$user" -path $log
  $getusermemberof = Get-ADUserMemberOf -User $user -Group
  $Adgroup #checking if user si already member
  if($getusermemberof -eq $true){ #if users is already mebe
  rjust write it to log
    Write-Log -Message "$user is already member of $Adgroup"
    -path $log
  }
  else{
    Write-Log -Message "ADD $user to $Adgroup" -path $log
    Add-ADGroupMember -identity $Adgroup -members $user
    if($error){ #error checking, if error occurs add in log
      Write-Log -Message "Error - ADD $user to $Adgroup" -path
      $log
     $error.clear() # clearing the error as it has already been
     captuire for this iteration
    }
```

```
    else{
      Write-Log -Message "Success - ADD $user to $Adgroup"
      -path $log
    }
  }
}
```

Step 5: Recycle logs or clean up the sessions (see Listing 10-1).

Listing 10-1. Cheat Code Script Template Example

```
#####################Recycle logs##############################
Set-Recyclelogs -foldername "logs" -limit $logrecyclelimit
-Confirm:$false

Write-Log -Message "Script Finished" -path $log

Glue is all together to form a nice script as shared in
Listing 10.1
<#
    .NOTES
    ===========================================================
    Created with:     ISE
    Created on:       9/6/2021 1:46 PM
    Created by:       Vikas Sukhija
    Organization:
    Filename:         ADDUserstoGroupfromText.ps1
    ===========================================================
    .DESCRIPTION
    This will add the users from text file to AD group
#>
###############Import modules and
functions####################
```

```
import-module vsadmin
import-module Activedirectory

###############Add logs and
variables#############################
$log = Write-Log -Name "ADDUser2Group-Log" -folder "logs" -Ext "log"
$users = get-content "c:\temp\users.txt"

$Adgroup = "ADgroup1"

$logrecyclelimit = "60" #to recycle the logs after 60 days
####################################################################
#
Write-Log -Message "Start..............script" -path $log

$users | foreach-object{
  $user = $_.trim() #triming fro any whitespace
  Write-Log -Message "Processing..........$user" -path $log
  $getusermemberof = Get-ADUserMemberOf -User $user -Group
  $Adgroup #checking if user si already member
  if($getusermemberof -eq $true){ #if users is already mebe
  rjust write it to log
    Write-Log -Message "$user is already member of $Adgroup"
    -path $log
  }
  else{
    Write-Log -Message "ADD $user to $Adgroup" -path $log
    Add-ADGroupMember -identity $Adgroup -member $user
    if($error){ #error checking, if error occurs add in log
      Write-Log -Message "Error - ADD $user to $Adgroup" -path
      $log
```

```
        $error.clear() # clearing the error as it has already
        been captuire for this iteration
      }
      else{
        Write-Log -Message "Success - ADD $user to $Adgroup"
        -path $log
      }
    }
}
#####################Recycle logs##############################
Set-Recyclelogs -foldername "logs" -limit $logrecyclelimit
-Confirm:$false
Write-Log -Message "Script Finished" -path $log
```

Let's run this cheat code by changing the variable adgroup and adding users in the text file as per the production environment. See Figure 10-3.

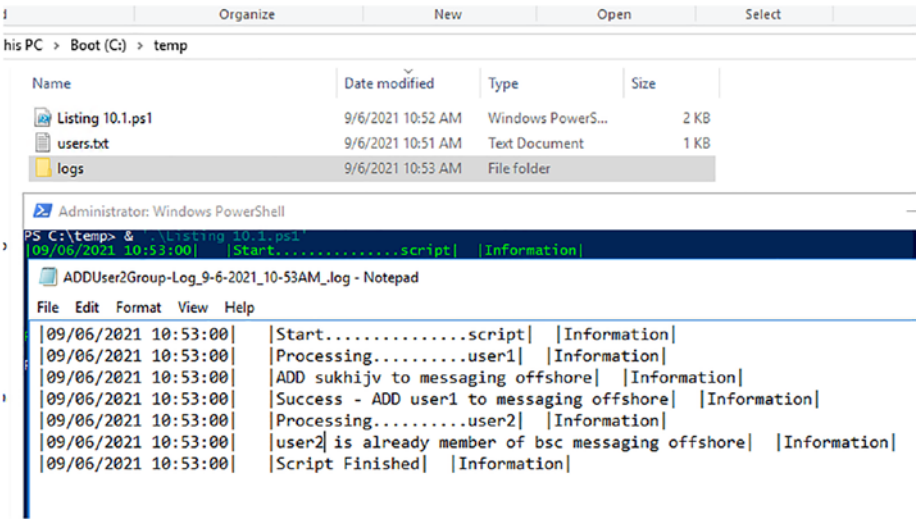


Figure 10-3. *Showing execution of the script ADDUserstoGroup FromText.ps1*

In a similar fashion, you can create multiple scripts for different production uses. I have shared hundreds of scripts with the community over the past 10 years, which you can access via the following link and modify as per your needs. The majority of scripts share the same principles described in this book: https://techwizard.cloud/downloads/.

Product Examples (Daily Use)

In this section, I share snippets that you can use as-is or combine in your scripts for daily admin tasks. Due to my love for Exchange, I use Microsoft Exchange ☺. Here are Exchange Script excerpts that you can use on a day-to-day basis.

Microsoft Exchange

Clean Database So That Mailboxes Appear in a Disconnected State

```
Get-MailboxServer | Get-MailboxDatabase | Clean-MailboxDatabase
```

Find Disconnected Mailboxes

```
Get-ExchangeServer | Where-Object {$_.IsMailboxServer -eq
$true} | ForEach-Object { Get-MailboxStatistics -Server $_.Name
| Where-Object {$_.DisconnectDate -notlike ''}}
```

Extract Message Accept From

```
Get-distributiongroup "dl name" | foreach {
$_.AcceptMessagesonlyFrom} | add-content  "c:/output/abc.txt"
```

Active Sync Stats

```
Get-CASMailbox -ResultSize unlimited | where {$_.
ActiveSyncEnabled -eq "true"} | ForEach-Object {Get-
ActiveSyncDeviceStatistics -Mailbox:$_.identity} | select
Devicetype, DeviceID,DeviceUserAgent, FirstSyncTime,
LastSuccessSync, Identity, DeviceModel, DeviceFriendlyName,
DeviceOS | Export-Csv c:\activesync.csv
```

Message Tracking

```
Get-transportserver | Get-MessageTrackingLog -Start "03/09/2015
00:00:00 AM" -End "03/09/2015 11:59:59 PM" -sender  "vikas@
lab.com" -resultsize unlimited  | select Timestamp,clientip,
ClientHostname,ServerIp,ServerHostname,sender,EventId,Messa
geSubject, TotalBytes , SourceContext,ConnectorId,Source ,
InternalMessageId , MessageId ,@{Name="Recipents";Expressi
on={$_.recipients}} | export-csv c:\track.csv
```

Search Mailbox/Delete Messages

Search only:

```
import-csv c:\tmp\messagesubject.csv | foreach {Search-Mailbox
$_.alias -SearchQuery subject:"Test SUbject" -TargetMailbox
"Exmontest" -TargetFolder "Logs" -LogOnly -LogLevel Full} >c:\
tmp\output.txt
```

Delete:

```
import-csv c:\tmp\messagesubject.csv | foreach {Search-Mailbox
$_.alias -SearchQuery subject:"Test Schedule" -DeleteContent
-force} >c:\tmp\output.txt
```

Delete and log:

```
import-csv c:\tmp\messagesubject.csv | foreach
{Search-Mailbox $_.alias  -SearchQuery Subject:"test
Story",Received:>'5/23/2018'  -TargetMailbox "Exmontest"
-TargetFolder "Logs"  -deletecontent -force} >c:\tmp\testlog-
23-29-left.txt
```

Exchange Quota Report

This example is found under Export-CSV as well.

```
#format Date
$date = get-date -format d
$date = $date.ToString().Replace("/", "-")
$output = ".\" + "QuotaReport_" + $date + "_.csv"
Collection = @()
Get-Mailbox -ResultSize Unlimited | foreach-object{
$st = get-mailboxstatistics $_.identity
$TotalSize = $st.TotalItemSize.Value.ToMB()
```

```
$user = get-user $_.identity
$mbxr = "" | select DisplayName,Alias,RecipientType,TotalItem
SizeinMB, QuotaStatus,
UseDatabaseQuotaDefaults,IssueWarningQuota,ProhibitSendQuota,
ProhibitSendReceiveQuota,
Itemcount, Email,ServerName,Company,Hidden, OrganizationalUnit,
RecipientTypeDetails,UserAccountControl,Exchangeversion

$mbxr.DisplayName = $_.DisplayName
$mbxr.Alias = $_.Alias
$mbxr.RecipientType = $user.RecipientType
$mbxr.TotalItemSizeinMB = $TotalSize
$mbxr.QuotaStatus = $st.StorageLimitStatus
$mbxr.UseDatabaseQuotaDefaults = $_.UseDatabaseQuotaDefaults
$mbxr.IssueWarningQuota = $_.IssueWarningQuota.Value
$mbxr.ProhibitSendQuota = $_.ProhibitSendQuota.Value
$mbxr.ProhibitSendReceiveQuota = $_.ProhibitSendReceiveQuota.Value
$mbxr.Itemcount = $st.Itemcount
$mbxr.Email = $_.PrimarySmtpAddress
$mbxr.ServerName = $st.ServerName
$mbxr.Company = $user.Company
$mbxr.Hidden = $_.HiddenFromAddressListsEnabled
$mbxr.RecipientTypeDetails = $_.RecipientTypeDetails
$mbxr.OrganizationalUnit = $_.OrganizationalUnit
$mbxr.UserAccountControl = $_.UserAccountControl
$mbxr.ExchangeVersion= $_.ExchangeVersion
$Collection += $mbxr
}
 #export the collection to csv , define the $output path
 accordingly
$Collection | export-csv $output
```

Set Quota

1GB mailbox limit (must have the $false included):

```
set-mailbox testmailbox -UseDatabaseQuotaDefaults $false
-IssueWarningQuota 997376KB -ProhibitSendQuota 1048576KB
-ProhibitSendReceiveQuota 4194304KB
```

2GB mailbox limit (must have the $false included):

```
set-mailbox "testmailbox" -UseDatabaseQuotaDefaults
$false -IssueWarningQuota 1.75GB -ProhibitSendQuota 2GB
-ProhibitSendReceiveQuota 4GB
```

3GB mailbox limit (must have the $false included):

```
set-mailbox "testmailbox" -UseDatabaseQuotaDefaults
$false -IssueWarningQuota 2.75GB -ProhibitSendQuota 3GB
-ProhibitSendReceiveQuota 5GB
```

Active Directory

Active Directory is the lifeline of every Microsoft product. By using PowerShell you can automate various AD components. Thankfully, Microsoft created a native Active Directory module for this job.

The following are methods that you can use for Active Directory scripting through PowerShell:

- Active Directory Module
- Quest Management Shell for Active Directory
- ADSI (out of scope for this book)

My favorite in the past was the Quest Management Shell followed by the Microsoft Active Directory Module. The Quest Shell is free and can be

downloaded. But Microsoft has added updates, so it is at par or better now than Quest, in my opinion.

One more reason for the AD module to take priority in my mind is that the Quest AD module is no longer free, you can still find the old version. I found it at the link below (if you want to use it in production, do cross check if there is licensing involved).

I encourage you to use the Microsoft Active directory module but there are still many admins or organizations using Quest, either freely or they have bought it. Download the Quest AD free version (1.5.1) from this link (found via Google): www.powershelladmin.com/wiki/Quest_ActiveRoles_Management_Shell_Download. See Figure 10-4.

Figure 10-4. *Showing the Quest AD module*

Exporting Group Members

Just a single line of code will work.

Using Quest:

```
Get-QADGroupMember "group Name" | select Name, Type | Export-
Csv .\members.csv
```

Using the AD module:

```
Get-ADGroup -identity "group Name"  -Properties member |
Select-Object -ExpandProperty member | Get-ADUser -Properties
DisplayName,Samaccountname,mail,employeeid | export-csv c:\
exportgroup.csv -notypeinfo
```

Setting Values for AD Attributes

Here is the example code that can be used to set AD attributes.

Using Quest:

```
Set-QADUser -identity samaccountname -ObjectAttributes
@{extensionattribute10 = "IntuneCommCompleted"}
```

Using the AD module:

```
Set-ADUser -identity samaccountanme -replace
@{"extensionattribute10" = "IntuneCommCompleted"}
```

Exporting Active Directory Attributes

This example is for calling Excel as well as using Quest ☺:

```
# call excel for writing the results
$objExcel = new-object -comobject excel.application
$workbook = $objExcel.Workbooks.Add()
$worksheet=$workbook.ActiveSheet
$objExcel.Visible = $False # true or false to set as visible on
screen or not
$cells=$worksheet.Cells
# define top level cell
$cells.item(1,1)="UserId"
$cells.item(1,2)="FirstName"
$cells.item(1,3)="LastName"
$cells.item(1,4)="Employeeid"
$cells.item(1,5)="email"
$cells.item(1,6)="Office"
$cells.item(1,7)="Department"
$cells.item(1,8)="Title"
$cells.item(1,9)="Company"
```

```
$cells.item(1,10)="City"
$cells.item(1,11)="State"
$cells.item(1,12)="Country"
#intitialize row out of the loop
$row = 2
#import quest management Shell
if ( (Get-PSSnapin -Name Quest.ActiveRoles.ADManagement
-ErrorAction SilentlyContinue) -eq $null )
{
    Add-PsSnapin Quest.ActiveRoles.ADManagement
}
$data = get-qaduser -IncludedProperties "CO",
"extensionattribute1" #-sizelimit 0
#loop thru users
foreach ($i in $data){
#initialize column within the loop so that it always loop back
to column 1
$col = 1
$userid=$i.Name
$FisrtName=$i.givenName
$LastName=$i.sn
$Employeeid=$i.extensionattribute1
$email=$i.PrimarySMTPAddress
$office=$i.Office
$Department=$i.Department
$Title=$i.Title
$Company=$i.Company
$City=$i.l
$state=$i.st
$Country=$i.CO
Write-host "Processing.............................$userid"
```

```
$cells.item($row,$col) = $userid
$col++
$cells.item($row,$col) = $FisrtName
$col++
$cells.item($row,$col) = $LastName
$col++
$cells.item($row,$col) = $Employeeid
$col++
$cells.item($row,$col) = $email
$col++
$cells.item($row,$col) = $office
$col++
$cells.item($row,$col) = $Department
$col++
$cells.item($row,$col) = $Title
$col++
$cells.item($row,$col) = $Company
$col++
$cells.item($row,$col) = $City
$col++
$cells.item($row,$col) = $state
$col++
$cells.item($row,$col) = $Country
$col++
$row++}
#formatting excel
$range = $objExcel.Range("A2").CurrentRegion
$range.ColumnWidth = 30
$range.Borders.Color = 0
$range.Borders.Weight = 2
$range.Interior.ColorIndex = 37
```

```
$range.Font.Bold = $false
$range.HorizontalAlignment = 3
# Headings in Bold
$cells.item(1,1).font.bold=$True
$cells.item(1,2).font.bold=$True
$cells.item(1,3).font.bold=$True
$cells.item(1,4).font.bold=$True
$cells.item(1,5).font.bold=$True
$cells.item(1,6).font.bold=$True
$cells.item(1,7).font.bold=$True
$cells.item(1,8).font.bold=$True
$cells.item(1,9).font.bold=$True
$cells.item(1,10).font.bold=$True
$cells.item(1,11).font.bold=$True
$cells.item(1,12).font.bold=$True
#save the excel file
$filepath = "c:\exportAD.xlsx" #save the excel file
$workbook.saveas($filepath)
$workbook.close()
$objExcel.Quit()
```

Same example using the native Active Directory module:

```
# call excel for writing the results
$objExcel = new-object -comobject excel.application
$workbook = $objExcel.Workbooks.Add()
$worksheet=$workbook.ActiveSheet
$objExcel.Visible = $True # true or false to set as visible on
screen or not
$cells=$worksheet.Cells
# define top level cell
$cells.item(1,1)="UserId"
```

```
$cells.item(1,2)="FirstName"
$cells.item(1,3)="LastName"
$cells.item(1,4)="Employeeid"
$cells.item(1,5)="email"
$cells.item(1,6)="Office"
$cells.item(1,7)="Department"
$cells.item(1,8)="Title"
$cells.item(1,9)="Company"
$cells.item(1,10)="City"
$cells.item(1,11)="State"
$cells.item(1,12)="Country"
#intitialize row out of the loop
$row = 2
#import AD management Shell
Import-module Activedirectory
$data = Get-ADUser -Filter {Enabled -eq $True} -Properties
extensionattribute1,mail,physicalDeliveryOfficeName,Department,
title,Company,l,st,co -ResultSetSize 1000 #define the size
#loop thru users
foreach ($i in $data){
#initialize column within the loop so that it always loop back
to column 1
$col = 1
$userid=$i.Name
$FisrtName=$i.givenName
$LastName=$i.surname
$Employeeid=$i.extensionattribute1
$email=$i.mail
$office=$i.physicalDeliveryOfficeName
$Department=$i.Department
$Title=$i.Title
```

```
$Company=$i.Company
$City=$i.l
$state=$i.st
$Country=$i.CO
Write-host "Processing.....................................$userid"
$cells.item($row,$col) = $userid
$col++
$cells.item($row,$col) = $FisrtName
$col++
$cells.item($row,$col) = $LastName
$col++
$cells.item($row,$col) = $Employeeid
$col++
$cells.item($row,$col) = $email
$col++
$cells.item($row,$col) = $office
$col++
$cells.item($row,$col) = $Department
$col++
$cells.item($row,$col) = $Title
$col++
$cells.item($row,$col) = $Company
$col++
$cells.item($row,$col) = $City
$col++
$cells.item($row,$col) = $state
$col++
$cells.item($row,$col) = $Country
$col++
$row++}
#formatting excel
```

```
$range = $objExcel.Range("A2").CurrentRegion
$range.ColumnWidth = 30
$range.Borders.Color = 0
$range.Borders.Weight = 2
$range.Interior.ColorIndex = 37
$range.Font.Bold = $false
$range.HorizontalAlignment = 3
# Headings in Bold
$cells.item(1,1).font.bold=$True
$cells.item(1,2).font.bold=$True
$cells.item(1,3).font.bold=$True
$cells.item(1,4).font.bold=$True
$cells.item(1,5).font.bold=$True
$cells.item(1,6).font.bold=$True
$cells.item(1,7).font.bold=$True
$cells.item(1,8).font.bold=$True
$cells.item(1,9).font.bold=$True
$cells.item(1,10).font.bold=$True
$cells.item(1,11).font.bold=$True
$cells.item(1,12).font.bold=$True
#save the excel file
$filepath = "c:\exportAD.xlsx" #save the excel file
$workbook.saveas($filepath)
$workbook.close()
$objExcel.Quit()
```

Adding Members to the Group from a Text File

Using the Quest Management Shell:

```
$users = Get-Content C:\Users.txt   # samccountnames of users
in text file
```

```
$groupname = "Group Name"
$users | ForEach-Object{
$user = $_
Write-host "adding $user to $groupname" -foregroundcolor green
Add-QADGroupMember -Identity $groupname -Member $user
}
```

Similarly, in the native Active Directory module:

```
$users = Get-Content C:\Users.txt    # samccountnames of users
in text file
$groupname = "Group Name"
$users | ForEach-Object{
$user = $_
Write-host "adding $user to $groupname" -foregroundcolor green
Add-ADGroupMember -id $groupname -members $user
}
```

Removing Members of the Group From a Text File

Using the Quest Management Shell:

```
$users = Get-Content C:\Users.txt   # samccountnames of users
in text file
$groupname = "Group Name"
$users | ForEach-Object{
$user = $_
Write-host "adding $user to $groupname" -foregroundcolor green
Remove-QADGroupMember -Identity $groupname -Member $user
-confirm:$false
}
```

Similarly, using the native Active Directory module:

```
$users = Get-Content C:\Users.txt    # samccountnames of users
in text file
$groupname = "Group Name"
$users | ForEach-Object{
$user = $_
Write-host "adding $user to $groupname" -foregroundcolor green
Remove-ADGroupMember -id $groupname -members $user
-confirm:$false
}
```

Office 365

Office 365 is everywhere so connecting is important in day-to-day activities for admins. You can use `vsadmin` or separate functions.

Operations: https://techwizard.cloud/2016/12/18/all-in-one-office-365-powershell-connect/

- `LaunchEOL/RemoveEOL` (Exchange Online)
- `LaunchSOL/RemoveSOL` (Skype online)
- `LaunchSPO/RemoveSPO` (SharePoint online)
- `LaunchCOL/RemoveCOL` (Security and Compliance)
- `LaunchMSOL/RemoveMSOL` (MS Online Azure Active Directory)

```
##################Exchange Modern Online##################
Function LaunchEOL {

  [CmdletBinding()]
  param
```

```
  (
    [Parameter(Mandatory = $false)]
    $Credential
  )
  Import-Module ExchangeOnlineManagement -Prefix "EOL"
  Connect-ExchangeOnline -Prefix "EOL" -Credential $Credential
  -ShowBanner:$false
}

Function RemoveEOL {
  Disconnect-ExchangeOnline -Confirm:$false
}

#############Skype Online#######################################
function LaunchSOL
{
  param
  (
    [Parameter(Mandatory = $true)]
    $Domain,
    [Parameter(Mandatory = $false)]
    $Credential
  )
  Write-Host -Object "Enter Skype Online Credentials"
  -ForegroundColor Green
  $dommicrosoft = $domain + ".onmicrosoft.com"
  $CSSession = New-CsOnlineSession -Credential $Credential
  -OverrideAdminDomain $dommicrosoft
  Import-Module (Import-PSSession -Session $CSSession
  -AllowClobber) -Prefix SOL  -Global
} #Function LaunchSOL

Function RemoveSOL
```

```
{
  $Session = Get-PSSession | Where-Object -FilterScript {
  $_.ComputerName -like "*.online.lync.com" }
  Remove-PSSession $Session
} #Function RemoveSOL

############Sharepoint Online#################################
function LaunchSPO
{
  param
  (
    [Parameter(Mandatory = $true)]
    $orgName,
    [Parameter(Mandatory = $false)]
    $Credential
  )
  Write-Host "Enter Sharepoint Online Credentials"
  -ForegroundColor Green
  Connect-SPOService -Url "https://$orgName-admin.sharepoint.
  com" -Credential $Credential
} #LaunchSPO

Function RemoveSPO
{
  disconnect-sposervice
} #RemoveSPO

####Secuirty and
Compliance#####################################
Function LaunchCOL {
  [CmdletBinding()]
  param
  (
```

```
    [Parameter(Mandatory = $false)]
    $Credential
  )
  Import-Module ExchangeOnlineManagement
  Connect-IPPSSession -Credential $Credential
  $s=Get-PSSession | where {$_.ComputerName -like "*compliance.
  protection.outlook.com"}
  Import-Module (Import-PSSession -Session $s  -AllowClobber)
  -Prefix col -Global
}

Function RemoveCOL {
  Disconnect-ExchangeOnline -Confirm:$false
}

################################Msonline#####################
function LaunchMSOL {

  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $false)]
    $Credential
  )
  import-module msonline
  Write-Host "Enter MS Online Credentials" -ForegroundColor Green
  Connect-MsolService -Credential $Credential
}

Function RemoveMSOL {
```

```
  Write-host "Close Powershell Window - No disconnect
  available" -ForegroundColor yellow
}
#########################################################
```

Exchange Online Mailbox Report

Now use the above function to launch the Exchange Online shell. In PowerShell, type `LaunchEOL` and supply the Exchange Online admin userid/password. Once you are connected to Exchange Online, run the following command to extract a mailboxes report from Office 365, which you can see in Figure 10-5:

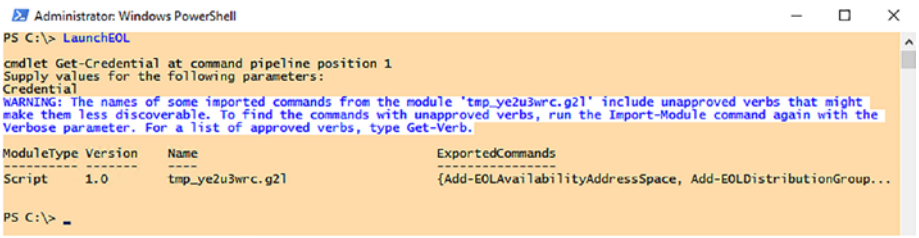


Figure 10-5. *Showing the connection to the Exchange shell*

```
Get-EOLMailbox -ResultSize unlimited | Select Name,Recipient
TypeDetails,PrimarySMTPAddress,UserPrincipalName,litigationhold
enabled,LitigationHoldDuration,PersistedCapabilities,Retention
HoldEnabled,RetentionPolicy,RetainDeletedItemsFor,ArchiveName,
Archivestatus,ProhibitSendQuota,ProhibitSendReceiveQuota,MaxSend
Size,MaxReceiveSize,AuditEnabled | export-csv c:\auditmbx.csv
-notypeinfo
```

If you have a large tenant, use this code instead as this will not throttle easily even with more than 50,000 users:

```
$allmbx=Invoke-Command -Session (Get-PSSession | Where-
Object{$_.computerName -eq "outlook.office365.com"})
-scriptblock {Get-Mailbox -ResultSize unlimited | Select-object
Name,RecipientTypeDetails, PrimarySMTPaddress,UserPrincipalname,
AuditEnabled,litigationholdenabled,LitigationHoldDuration,
PersistedCapabilities,RetentionHoldEnabled,RetentionPolicy,
RetainDeletedItemsFor,ArchiveName,Archivestatus,ArchiveGuid,
ProhibitSendQuota,ProhibitSendReceiveQuota,MaxSendSize,Max
ReceiveSize,WhenMailboxCreated,WhenCreated,HiddenFromAddress
ListsEnabled }
 $allmbx | export-csv c:\data\auditmbx.csv -notypeinfo
```

Exchange Online Message Tracking

In Exchange Online, extracting message tracking is not the same as it is in Exchange on-premise, because if the results are more in number, then it cannot be extracted using a result size unlimited parameter. The following is a small script that will do the trick:

```
$index = 1
while ($index -le 1001)
{
Get-EOLMessageTrace -SenderAddress "VikasS@techWizard.cloud"
-StartDate 09/20/2019 -EndDate 09/25/2019 -PageSize 5000 -Page
$index | export-csv c:\messagetracking.csv -Append
$index ++
sleep 5
}
```

Searching a Unified Log

Office 365 uses unified audit logging and you can audit all of the activities using the Exchange Online shell (whether it is SharePoint Online or Teams or any other product within Office 365). Here is the link for more details:

https://docs.microsoft.com/en-us/microsoft-365/compliance/search-the-audit-log-in-security-and-compliance?WT.mc_id=M365-MVP-5001317

Example of extracting Microsoft Teams activity:

```
Search-EOLUnifiedAuditLog -StartDate 1/8/2019 -EndDate
4/7/2019 -RecordType MicrosoftTeams -UserIds VikasS@sycloudpro.
com  -ResultSize:5000 |export-csv c:\VikasS.csv -notypeinfo
```

Example of extracting Exchange mailbox audit activity:

```
Search-EOLUnifiedAuditLog -StartDate 10/24/2019 -EndDate
10/25/2019 -UserIds VikasS@syscloudpro.com -recordtype "Exch
angeItemGroup","ExchangeItem","ExchangeAggregatedOperation"
-ResultSize:5000 |export-csv c:\VikasS.csv -notypeinfo
```

Example of adding or removing a role member:

```
Search-EOLUnifiedAuditLog -StartDate 4/16/2019 -EndDate
7/15/2019 -UserIds VikasS@syscloudpro.com -operations "Add
role member to role" -ResultSize:5000 |export-csv c:\VikasS.csv
-notypeinfo
```

Azure AD

I have covered practical examples of Active Directory but in today's world Azure AD is becoming common so here are some example from the Azure AD world.

For Azure AD, you need to use `connect-AzureAD` first to connect, and for MS Online you can use `Connect-MsolService`. Once connected, you will be able to use the following examples by updating the variables.

Adding Users to an Azure AD Group From a Text File of UPN

```
$group1 = "93345231-7454-4629-943b-e4245426bf" #
Get-Content C:\users.txt | ForEach-Object{$user=$_.
trim();$user;$upn= $user
$getazureaduser = Get-AzureADUser -Filter "userprincipalname eq
'$($upn)'"
 Add-AzureADGroupMember -ObjectId $group1  -RefObjectId
 $getazureaduser.ObjectId
}
```

Removing Users in an Azure AD Group from a Text File of UPN

```
$group1 = "93345231-7454-4629-943b-e4245426bf" #
Get-Content C:\users.txt | ForEach-Object{$user=$_.
trim();$user;$upn= $user
$getazureaduser = Get-AzureADUser -Filter "userprincipalname eq
'$($upn)'"
Remove-AzureADGroupMember -ObjectId $group1  -MemberId
$getazureaduser.ObjectId
}
```

Checking If a User Is Already a Member of a Group

```
$group1 = "93345231-7454-4629-943b-e4245426bf" #
$getazmembership = Get-AzureADUserMembership  -ObjectId
"UserObjectId"
if($getazmembership.objectId -contains $group1){
write-host  "User is already member of the group group1"
}
```

Adding Administrators to a Role

```
Get-MsolRole | Sort Name | Select Name,Description #check role name
$roleName = "Lync Service Administrator"
Get-content c:\users.txt | foreach-object{$_;
Add-MsolRoleMember -RoleMemberEmailAddress $_ -RoleName
$roleName
}
```

Checking for Azure AD User Provisioning Errors

```
Get-MsolUser -HasErrorsOnly | ft
DisplayName,UserPrincipalName,@{Name="Error";Expression={($_.
errors[0].ErrorDetail.objecterrors.errorrecord.
ErrorDescription)}} -AutoSize
```

In a similar fashion, you can connect to any Microsoft product by checking their documentation. As for other Azure products, there is a command named `Connect-AzAccount` for a connection to Azure. Just make sure that the modules are installed on your machines for whichever product you want to connect to in the cloud.

Text/CSV File Operations

Remove the header line from a CSV file

Method 1:

```
Get-Content .\abc.csv | select -skip 1 | Set-Content .\abc1.csv
```

Method 2:

```
$a = import .\abc.csv
$a |ForEach-Object{
  $Con_string = $null
  $Con_string = $_.ID, $_.GrpName -join ','
  Write-Host $Con_string
  Add-Content .\abc6.csv $Con_string
}
```

Method 3 (avoids CRLF):

```
$text = [System.IO.File]::ReadAllText("$pwd\file.csv") -replace
'^[^\r\n]*\r?\n'
[System.IO.File]::WriteAllText("$pwd\newFile.csv", $text)
```

Method 4 (avoids CRLF):

```
$file = Get-Item .\example_test.csv
$reader = $file.OpenText()
# discard the first line
$null = $reader.ReadLine()
```

```
# Write the rest of the text to the new file
[System.IO.File]::WriteAllText("$pwd\newFile.csv", $reader.
ReadToEnd())
$reader.Close()
```

Adding a header line to a text file:
For example, you have list of employee IDs in a text file:

14562

67578

65888

```
$filep = "c:\file.txt"
$getNetworkID = Get-Content $filep | where { $_ -ne "" }
@("Employeeid") +  $getNetworkID | Set-Content  $filep -Force
#add emplyeeidheader
```

Regex

There are situations where you need to use regex for performing certain match operations inside your scripts.

Tip You can use https://regex101.com/ to test any regex before using it.

This is how you use it in PowerShell and it will be used mainly with match operators. See Figures 10-6 and 10-7.

```
$regexemail = "^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$"
"sukhijav@techwizard.cloud" -match $regexemail
```

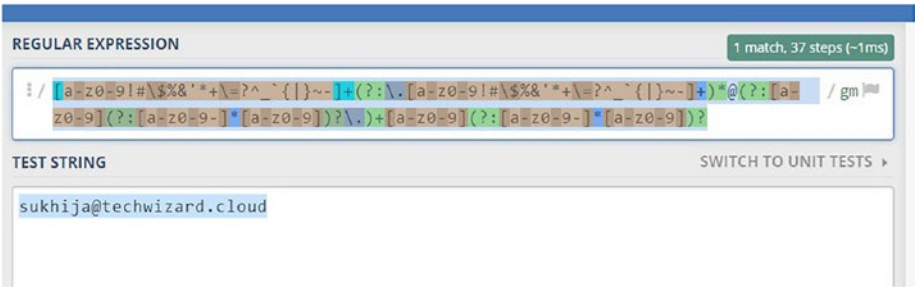


Figure 10-6. *Showing regular expression testing*

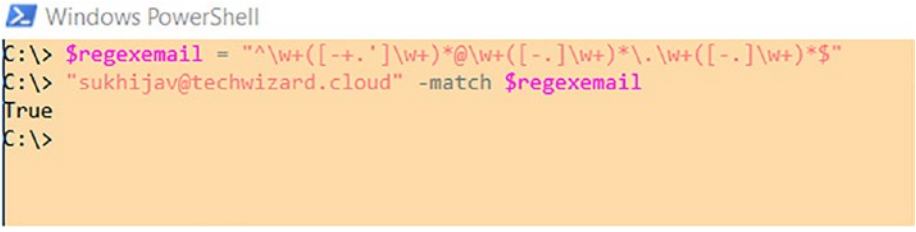


Figure 10-7. *Showing a regular expression operation in PowerShell*

Sno.	Regex Cheat	Comments
1	`Receipt_[0-9][0-9][0-9][0-9][0-9][0-9][0-9]\.doc`	Contains `Reciept_7didgit number.doc`
2	`(Tickets issued to)(.*)(for travel)`	Tickets issued to Vikas Sukhija for travel
3	`(.*)Aborted_payment_(.*)`	Tell Aborted_payment_(Y075958)
4	`(.*)(\([A-Z][0-9][0-9][0-9][0-9][0-9][0-9]\)`	(Y782714)
5	`(.*)[0-9]{2}[A-Z]{1}[0-9]{6}`	Critical_alert_-_36B881478
6	`(?<=V0)(.*)(?=a)`	V01234a
7	`^\d+$`	For finding an integer
8	`^0+$`	For finding an integer with 000000
9	`^\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$`	For email

Summary

This is the last chapter of this book and it was all about how you can combine different snippets and make a script that can do a bulk load of work. I have shared different product examples that can be used by system administrators in their daily work. More scripts and hundreds of examples are available at https://techwizard.cloud/downloads/.

Index

A, B

C

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/978-1-4842-7759-1

D

E, F, G

H

I, J, K

L

M, N

O

P, Q

R

S

T, U

V, W, X, Y, Z