PowerShell Fast Track

Hacks for Non-Coders

Second Edition

Vikas Sukhija

Apress®

PowerShell Fast Track

Hacks for Non-Coders

Second Edition

Vikas Sukhija

Apress®

PowerShell Fast Track: Hacks for Non-Coders, Second Edition

Vikas Sukhija
Waterloo, ON, Canada

ISBN-13 (pbk): 979-8-8688-1156-2 ISBN-13 (electronic): 979-8-8688-1157-9
https://doi.org/10.1007/979-8-8688-1157-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Desk Editor: Laura Berendson
Editorial Project Manager: Kripa Joseph

Cover image by Freepik (www.freepik.com)

Table of Contents

About the Author

Vikas Sukhija, widely recognized as *TechWizard*, brings over two decades of expertise in IT infrastructure, with a deep specialization in messaging, collaboration, and IT automation. Leveraging powerful tools like PowerShell, PowerApps, and Power Automate, he has successfully designed and implemented automation solutions across a range of platforms. Currently serving as Principal Architect at Boston Scientific, Vikas is a driving force behind modern IT automation frameworks and best practices. As a Microsoft MVP and seasoned blogger, he actively shares his knowledge with the global tech community, guiding enterprises of all sizes in architecting, automating, and streamlining their Microsoft 365 and Azure environments.

About the Technical Reviewer

Arun Sharma is the CEO of Suri Technologies, where he spearheads the cloud and AI business across multiple regions. With over 25 years of experience, Arun has developed a robust expertise in cloud technologies, including Microsoft Azure, AWS, and GSuite, as well as IoT, machine learning, microservices, and containerization.

His impressive career includes leadership roles such as AVP of Cloud Solutions at Click2Cloud, General Manager of Cloud and AI at Paytm and AliCloud, and Delivery Manager at Microsoft. He has also served as a Product Manager at Icertis and held various positions at Infosys and CMC. Throughout his career, Arun has successfully managed relationships and sales with medium and enterprise global clients, driving cloud consumption and consulting services.

Arun thrives on challenges in the Microsoft ecosystem, leveraging his deep domain knowledge in sectors such as banking, insurance, FMCG, government, retail, and agritech. He is actively involved in the tech community as an author of international research papers, a technical speaker, and a trainer. Recognized as a Microsoft Certified Trainer (MCT), Arun holds a Doctor of Business Administration, an MBA, and an MTech in Computer Science.

CHAPTER 1

PowerShell Basics

PowerShell has evolved significantly since the earlier editions of this book. With the introduction of PowerShell 7, we now have a powerful, cross-platform scripting language that runs on Windows, macOS, and Linux. This is a notable shift from the Windows-only constraint of PowerShell versions up to 5.

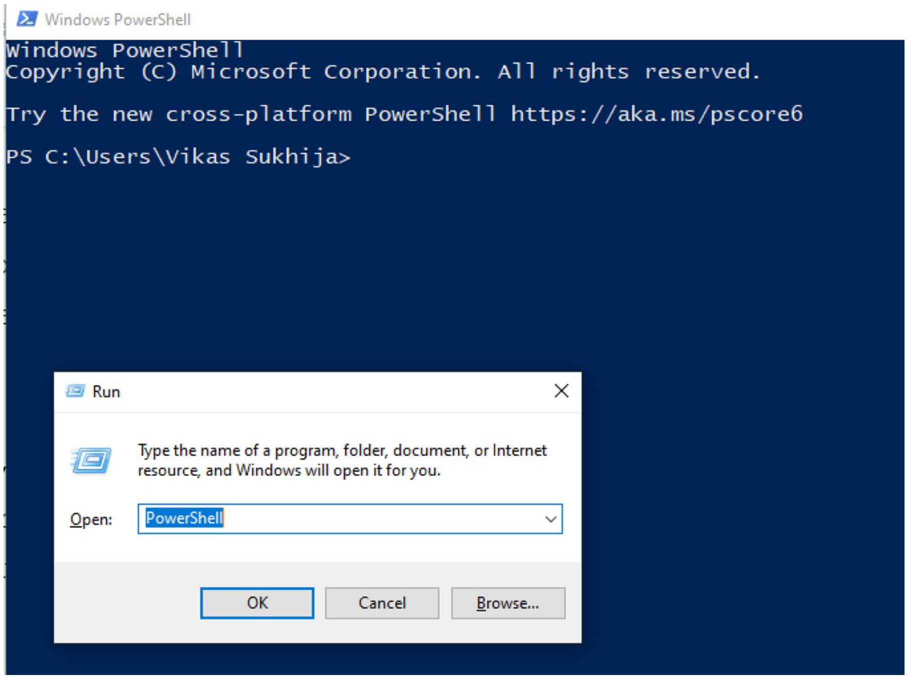
PowerShell 7 brings several advancements, including enhanced performance and a reduced memory footprint. The development focus has moved to PowerShell 7 and later versions, ensuring that new features and improvements are continually integrated into the platform.

However, PowerShell 5.1 is far from obsolete. It remains in use, with many developers continuing to create and maintain scripts in this version. Most scripts are compatible across both versions, though some may require minor adjustments to run smoothly in PowerShell 7.

To run PowerShell version 5, you use PowerShell as the keyword in the run command of Windows.

Refer Figure 1-1.

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_1



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Vikas Sukhija>
```

Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: PowerShell

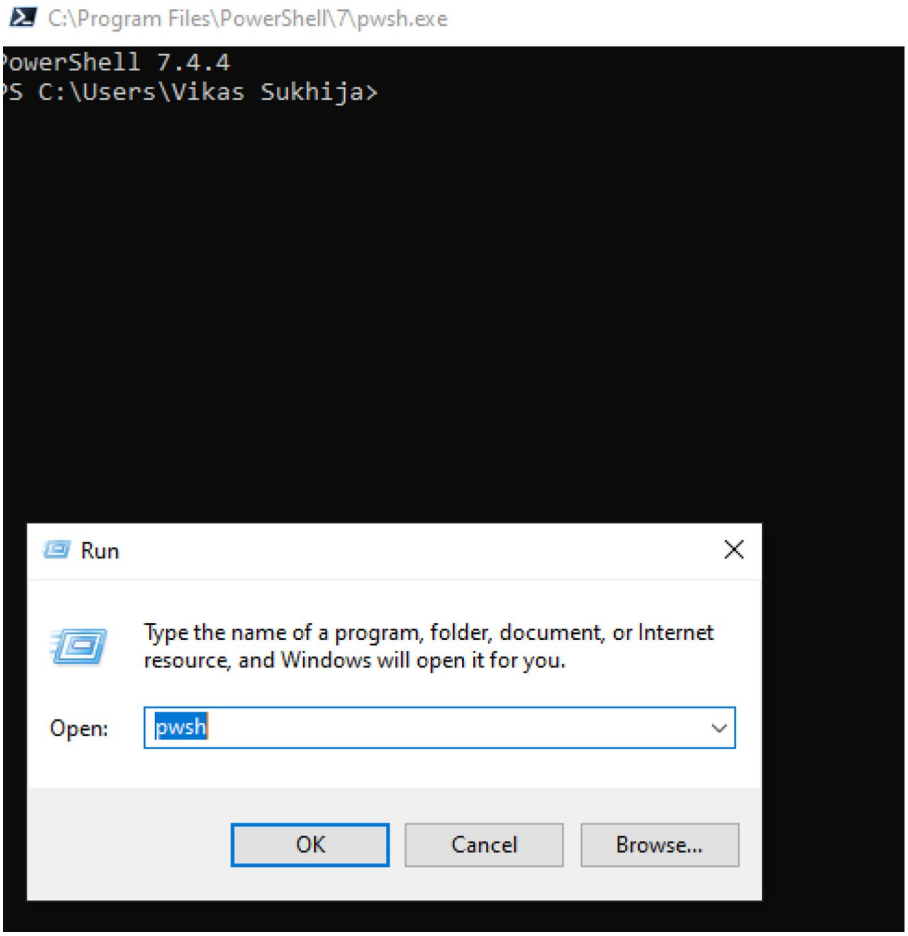
Figure 1-1. *Running Powershell 5*

To run PowerShell version 7, you use pwsh as the keyword in the run command of Windows.

Do not forget to install it first:

`https://learn.microsoft.com/en-us/shows/it-ops-talk/how-to-install-powershell-7`

Refer Figure 1-2.


C:\Program Files\PowerShell\7\pwsh.exe

```
PowerShell 7.4.4
PS C:\Users\Vikas Sukhija>
```

Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open: pwsh
OK
Cancel
Browse...

***Figure 1-2.** Running PowerShell 7*

Let's begin with the foundational elements of scripting: variables, loops, if/else statements, switches, and functions. These core components are the backbone of any scripting language, enabling you to craft scripts that range from the simplest tasks to the most complex automation.

I won't delve into the various versions of PowerShell or provide an extensive definition of what PowerShell is. For the sake of clarity, PowerShell is essentially a task automation solution comprising a command-line shell and a scripting language. This book isn't about the intricacies of the language or the platforms it supports, nor will it cover get and set commands in detail. Instead, it's designed to help you create scripts efficiently without needing an in-depth understanding of the underlying mechanics.

My goal is to equip you with the skills to develop scripts through a hands-on approach. This method has proven effective for many of my students, who have gradually mastered the language over time.

You don't need a programming background or coding expertise to benefit from this book. By following the practical approach laid out here, you'll quickly learn to write your own scripts and automate various IT systems and processes. This way, you can focus on achieving tangible results without getting bogged down by complex concepts.

Note All source code used in the book can be accessed by clicking **Download Source Code:** **`https://github.com/Apress/PowerShell-Fast-Track-Second-Edition`** (follow the listing numbers).

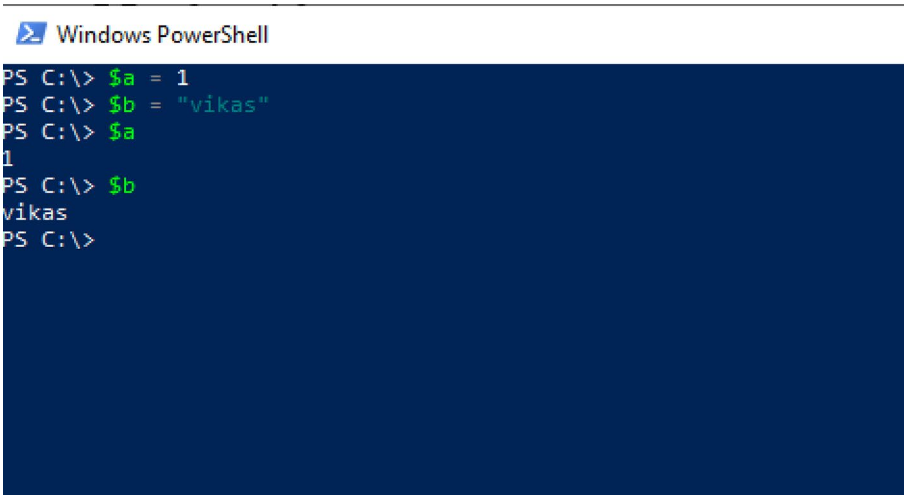
Variables and On-Screen Printing

First, let's cover the basics: variables and arrays. In PowerShell, every variable starts with a dollar sign ($). This helps distinguish variables from other elements in your script.

Here are a couple of examples:

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

When you type $a and $b, values will be displayed as shown in Figure 1-3.



```
PS C:\> $a = 1
PS C:\> $b = "vikas"
PS C:\> $a
1
PS C:\> $b
vikas
PS C:\>
```

Figure 1-3. *Variables in PowerShell*

Note I have not written 1 in quotes, but when I used string, quotes are utilized.

PowerShell autodetermines the first value as int.

Let us now print these variables on the screen. You can do that using write-host and Write-Output.

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

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

Write-host also has parameters ForegroundColor and BackgroundColor which you can use to change colors on print output.

```
Input: PS C:\> Write-host $a -foregroundcolor green
Output: 1
```

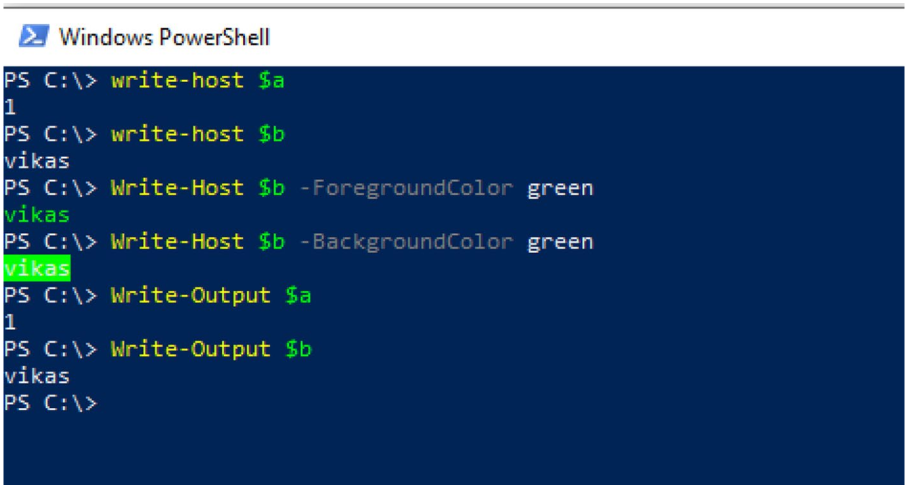
```
Input: PS C:\> Write-host $b -backgroundcolor green
Output: vikas
```

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

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

Tip Make it a rule to use quotes when assigning values to variables when you are working with strings, as shown above in the example.

Figure 1-4 shows the results.



Windows PowerShell

```
PS C:\> write-host $a
1
PS C:\> write-host $b
vikas
PS C:\> Write-Host $b -ForegroundColor green
vikas
PS C:\> Write-Host $b -BackgroundColor green
vikas
PS C:\> Write-Output $a
1
PS C:\> Write-Output $b
vikas
PS C:\>
```

Figure 1-4. *Using the Write-Host and Write-Output variable*

Arrays and array lists:

In PowerShell, arrays and array lists are two methods for storing collections of items, each with its own advantages. While both serve similar purposes, I recommend using array lists whenever possible due to their superior performance.

Arrays:

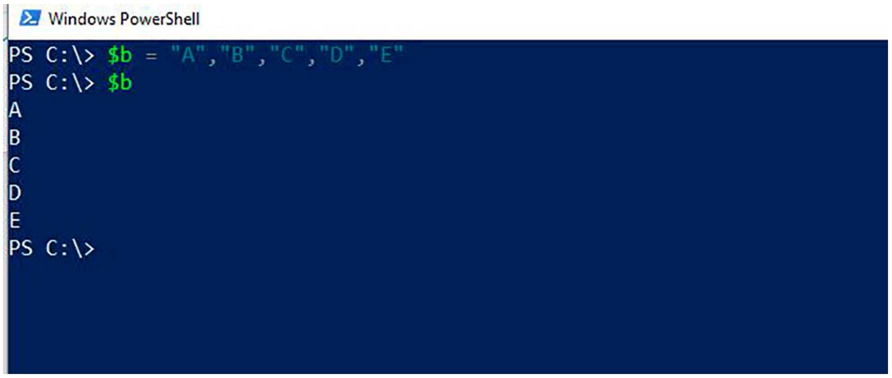
Arrays are fixed-size collections of items. When you create an array, its size is set, and you cannot change it without creating a new array. This behavior makes arrays less efficient when dealing with large datasets or when the size of the collection needs to change frequently. When you add an item to the array, it deletes the existing array and creates a new array which makes them very slow.

Below are different ways you can create the arrays in PowerShell. You can create them in the same way as you have created the variable, just need to separate the elements by a comma.

Here are some examples:

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

Refer Figure 1-5.



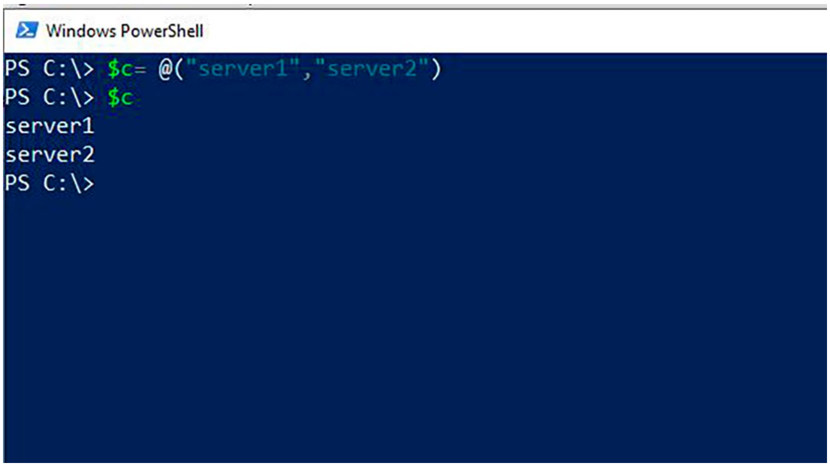
```
PS C:\> $b = "A","B","C","D","E"
PS C:\> $b
A
B
C
D
E
PS C:\>
```

Figure 1-5. *Array illustration*

Another approach to defining an array in PowerShell involves using the @() syntax. This method is intuitive because the syntax itself clearly indicates that you are creating an array.

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

This is shown in Figure 1-6



Windows PowerShell

```
PS C:\> $c= @("server1","server2")
PS C:\> $c
server1
server2
PS C:\>
```

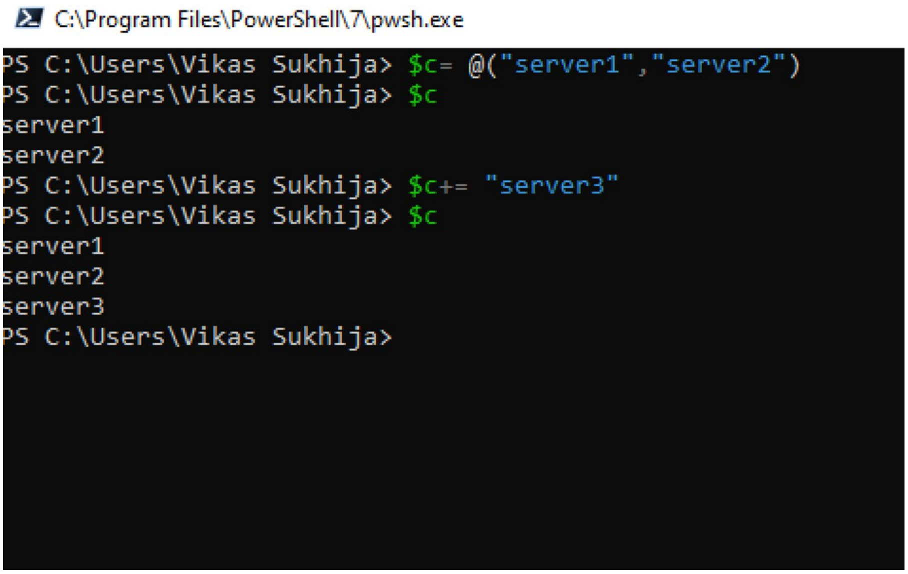
Figure 1-6. *Array syntax*

A dynamic array in PowerShell can be initialized with the @() syntax, which creates an empty array. This type of array is particularly useful when you need to add elements to it dynamically as shown in Figure 1-7

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

How to ADD element to an array:

```
$c= @("server1","server2")
$c+= "server3"
```



```
PS C:\Users\Vikas Sukhija> $c= @("server1","server2")
PS C:\Users\Vikas Sukhija> $c
server1
server2
PS C:\Users\Vikas Sukhija> $c+= "server3"
PS C:\Users\Vikas Sukhija> $c
server1
server2
server3
PS C:\Users\Vikas Sukhija>
```

Figure 1-7. *Add element to array*

Array lists:

Array list is type of collection you should always use as a thumb rule when you are scripting as it is performance rich when dataset is huge.

ArrayLists provide significant performance advantages over traditional arrays.

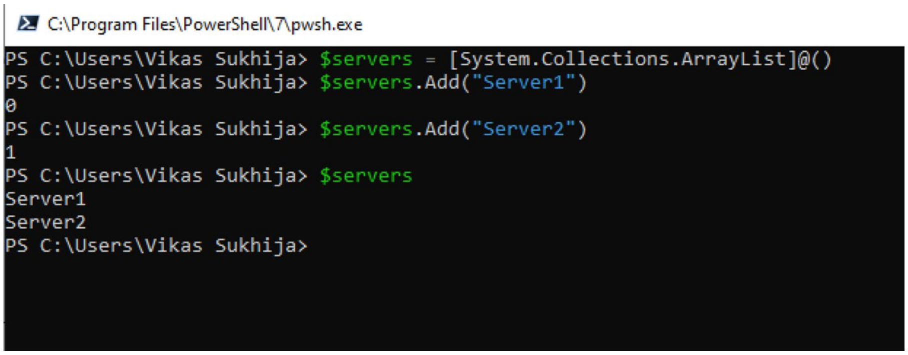
Syntax to create array list:

```
$servers = [System.Collections.ArrayList]@()
```

How to ADD element to an array list:

```
$servers.Add("Server1")
$servers.Add("Server2")
```

Note the Add keyword that is required for adding the element as shown in below Figure 1-8


C:\Program Files\PowerShell\7\pwsh.exe

```
PS C:\Users\Vikas Sukhija> $servers = [System.Collections.ArrayList]@()
PS C:\Users\Vikas Sukhija> $servers.Add("Server1")
0
PS C:\Users\Vikas Sukhija> $servers.Add("Server2")
1
PS C:\Users\Vikas Sukhija> $servers
Server1
Server2
PS C:\Users\Vikas Sukhija>
```

Figure 1-8. *Add element to array list*

If/Else

In scripting, conditional processing is fundamental. The if-else statement allows you to execute different code blocks based on whether a specified condition is true or false. This concept is universal across scripting languages and forms the basis of decision-making in your scripts.

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 1-9.

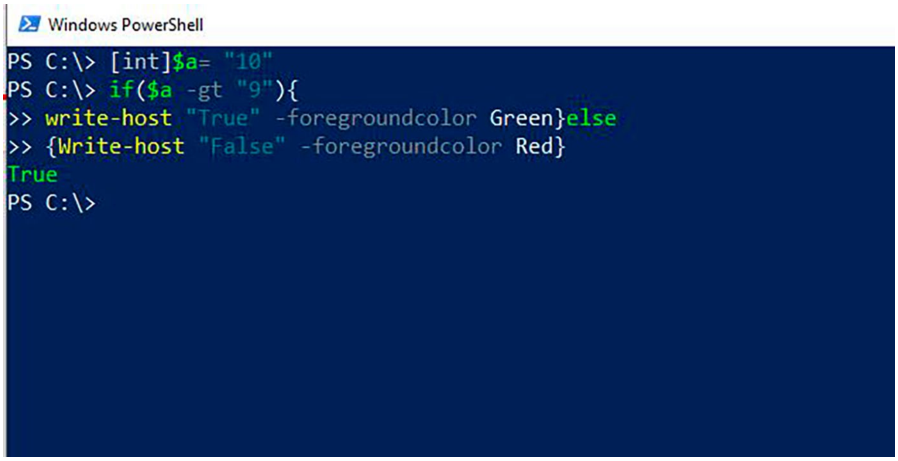
Note that `-gt` means greater than and `-lt` means less than. I will quickly go through them in the next subsection.

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
}
```

[int]in front of $a means integer. If you use a prefix before the variable, it means you have exclusively defined its type, like the above [int] has been prefixed. Setting prefix types is always better, but if you don't, PowerShell is intelligent enough to do it implicitly. These are called data types, and they include [string], [char], [int], [array], etc. Using type casting makes your code more predictable by explicitly defining the data type of a variable.



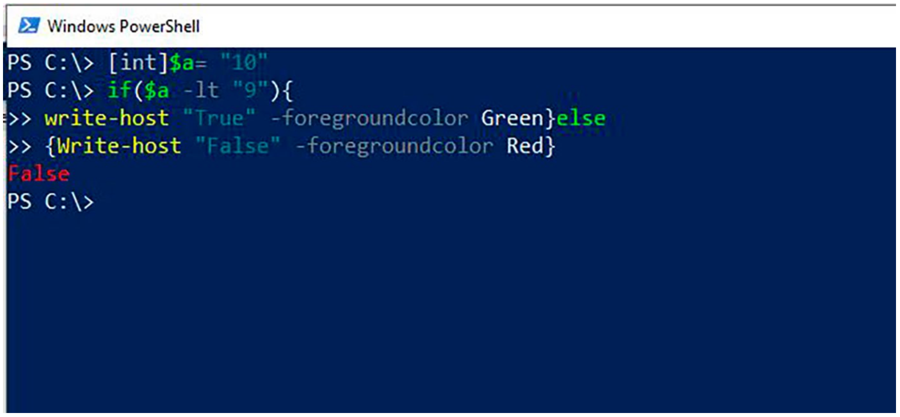
```
PS C:\> [int]$a= "10"
PS C:\> if($a -gt "9"){
>> write-host "True" -foregroundcolor Green}else
>> {Write-host "False" -foregroundcolor Red}
True
PS C:\>
```

Figure 1-9. *Showing -gt usage in if/else*

Listing 1-2 and Figure 1-10 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
}
```



```
PS C:\> [int]$a= "10"
PS C:\> if($a -lt "9"){
>> write-host "True" -foregroundcolor Green}else
>> {Write-host "False" -foregroundcolor Red}
False
PS C:\>
```

Figure 1-10. *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. These operators are crucial for performing comparisons and making decisions in your scripts.

`-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

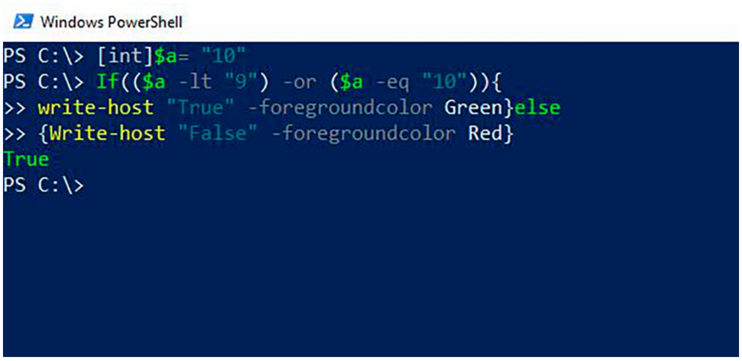
!: Logical NOT

Logical operators allow you to combine multiple conditions in your scripts. They are essential for more complex decision-making processes. In PowerShell, you can use logical operators to check if either or both of the several conditions are true.

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 1-11.

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}
```



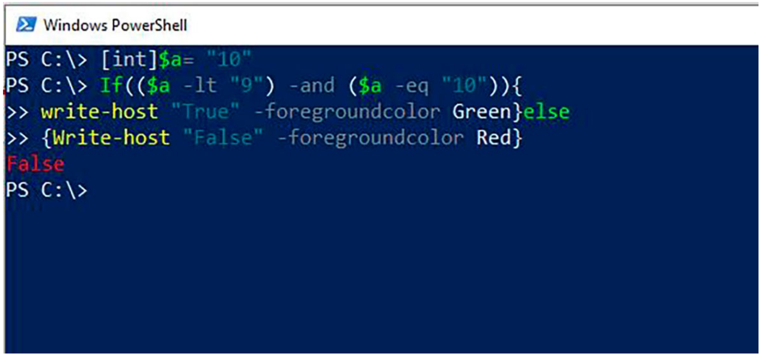
```
PS C:\> [int]$a= "10"
PS C:\> If(($a -lt "9") -or ($a -eq "10")){
>> write-host "True" -foregroundcolor Green}else
>> {Write-host "False" -foregroundcolor Red}
True
PS C:\>
```

Figure 1-11. *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 1-12.

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}
```



```
PS C:\> [int]$a= "10"
PS C:\> If(($a -lt "9") -and ($a -eq "10")){
>> write-host "True" -foregroundcolor Green}else
>> {Write-host "False" -foregroundcolor Red}
False
PS C:\>
```

Figure 1-12. *Showing logical -and operator*

The -not operator in PowerShell is used for negation, meaning it inverts the result of a condition. If a condition evaluates to true, -not makes it false, and vice versa.

While the -not operator is useful for certain scenarios, I typically avoid using it frequently. In my experience, it can lead to mistakes, especially if you're working quickly and not paying close attention. Misinterpreting the results can occur easily, making your script harder to debug.

For more straightforward or less error-prone alternatives, consider using positive conditions and combining them with logical operators where possible. This can help you avoid potential pitfalls and make your code more readable and maintainable.

Switch

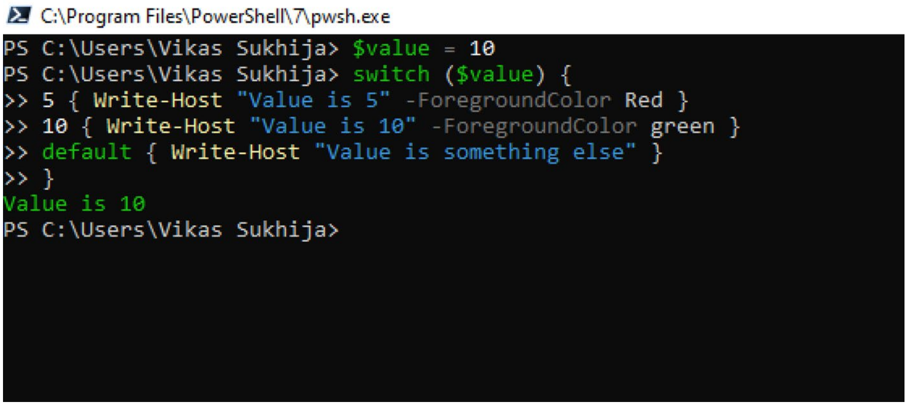
Switch case is another type of statement which you can utilize to handle multiple conditions based on value of single variable. When you are dealing with a single variable that can have multiple distinct values, making it easier to handle multiple cases cleanly. Avoid using if else in this case as otherwise your code will become huge and will look ugly and difficult to understand.

Syntax for switch case:

```
switch($variable){
case1 { # Code to execute if matches case1 }
case2 { # Code to execute if matches case2 }
case3 { # Code to execute if matches case3 }
default { #execute if none of the cases match }
}
```

Listing 1-5. Showing switch case

```
$value = 10
switch ($value) {
      5 { Write-Host "Value is 5" -ForegroundColor Red }
      10 { Write-Host "Value is 10" -ForegroundColor green }
      default { Write-Host "Value is something else" }
}
```



```
C:\Program Files\PowerShell\7\pwsh.exe
PS C:\Users\Vikas Sukhija> $value = 10
PS C:\Users\Vikas Sukhija> switch ($value) {
>> 5 { Write-Host "Value is 5" -ForegroundColor Red }
>> 10 { Write-Host "Value is 10" -ForegroundColor green }
>> default { Write-Host "Value is something else" }
>> }
Value is 10
PS C:\Users\Vikas Sukhija>
```

Figure 1-13. *Showing usage of SWITCH case*

Loops

In PowerShell, as in many other scripting languages, loops are essential for executing a block of code repeatedly. There are two primary types of loops, with others being variations or combinations of these.

For Loop and While Loop

For Loop

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

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

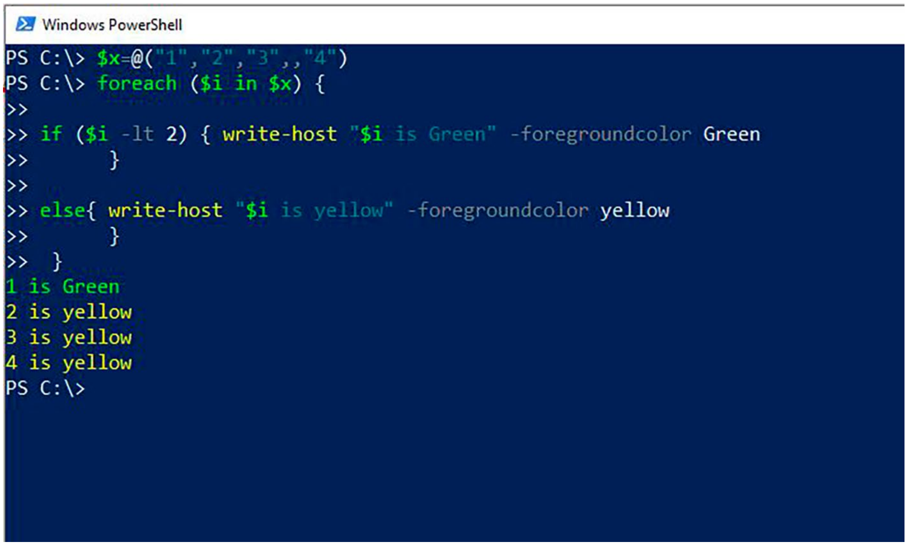
Let's differentiate between the three `for` loops by looking at the examples.

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

Note You can combine the `if else` and comparison operators in Listing 1-6. You can see the results in Figure 1-14.

Listing 1-6. 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
      }
 }
```



Windows PowerShell

```
PS C:\> $x=@("1","2","3",,"4")
PS C:\> foreach ($i in $x) {
>>
>> if ($i -lt 2) { write-host "$i is Green" -foregroundcolor Green
>>      }
>>
>> else{ write-host "$i is yellow" -foregroundcolor yellow
>>      }
>>  }
1 is Green
2 is yellow
3 is yellow
4 is yellow
PS C:\>
```

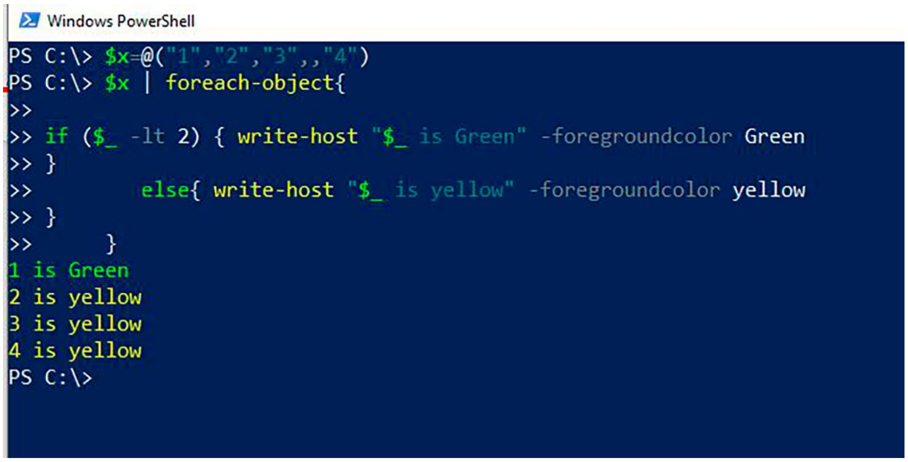
***Figure 1-14.** Showing a foreach loop*

`foreach-object`: You use a PIPE with the collection to achieve the same thing (see Listing 1-7 and Figure 1-15):

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

Listing 1-7. 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
}
     }
```



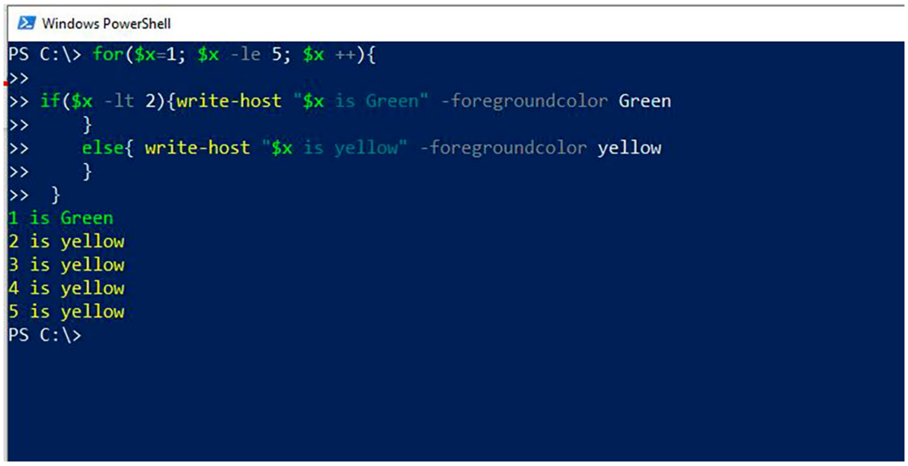
```
PS C:\> $x=@("1","2","3",,"4")
PS C:\> $x | foreach-object{
>>
>> if ($_ -lt 2) { write-host "$_ is Green" -foregroundcolor Green
>> }
>>         else{ write-host "$_ is yellow" -foregroundcolor yellow
>> }
>>     }
1 is Green
2 is yellow
3 is yellow
4 is yellow
PS C:\>
```

Figure 1-15. *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 1-8 and the results in Figure 1-16.

Listing 1-8. 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
    }
 }
```



```
PS C:\> 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
>>     }
>>  }
1 is Green
2 is yellow
3 is yellow
4 is yellow
5 is yellow
PS C:\>
```

Figure 1-16. *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:

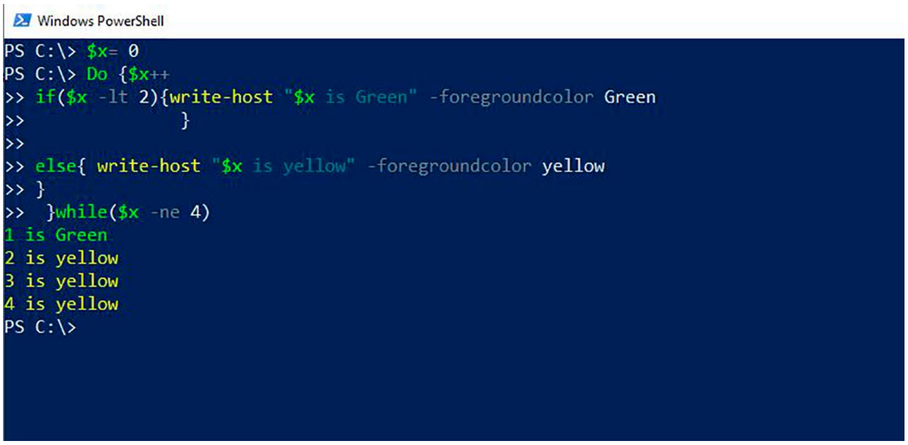
- `do-while`
- `while`

For `do-while`, you do something until some condition is met. In Listing 1-9, variable `x = 0`, and inside the variable, you increment its value until it is not equal to 4. See Figure 1-17 for the result.

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

Listing 1-9. 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)
```


Windows PowerShell

```
PS C:\> $x= 0
PS C:\> Do {$x++
>> if($x -lt 2){write-host "$x is Green" -foregroundcolor Green
>>               }
>>
>> else{ write-host "$x is yellow" -foregroundcolor yellow
>> }
>>  }while($x -ne 4)
1 is Green
2 is yellow
3 is yellow
4 is yellow
PS C:\>
```

Figure 1-17. *Showing a do-while loop*

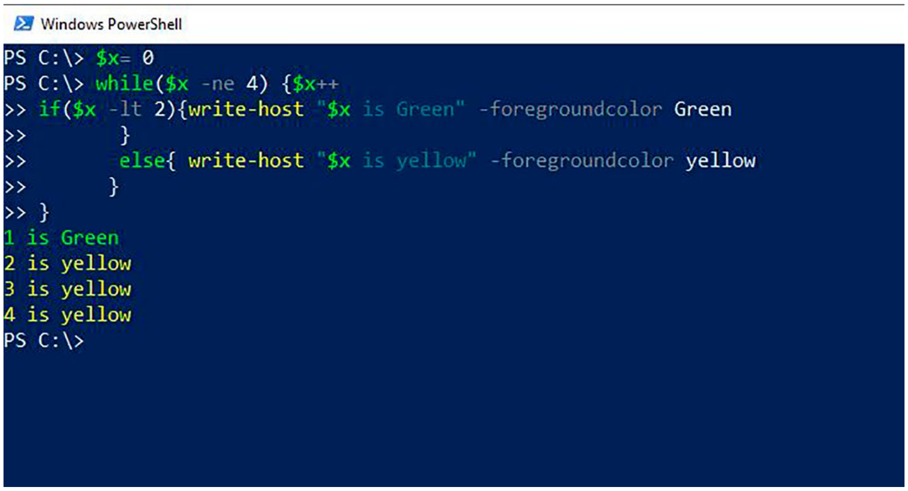
For while, you are also doing something until some condition is met. In Listing 1-10, variable x = 0, and inside the variable, you increment its value until it is not equal to 4.

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 1-10) checks the condition before the loop (iteration) but `do-while` does the checks after the execution. See Figure 1-18 for the result.

Listing 1-10. 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
      }
}
```



Windows PowerShell

```
PS C:\> $x= 0
PS C:\> while($x -ne 4) {$x++
>> if($x -lt 2){write-host "$x is Green" -foregroundcolor Green
>>        }
>>        else{ write-host "$x is yellow" -foregroundcolor yellow
>>       }
>> }
1 is Green
2 is yellow
3 is yellow
4 is yellow
PS C:\>
```

Figure 1-18. *Showing a while loop*

Functions

Functions are reusable blocks of code that you define once and can call from anywhere in your script. They help streamline your code by reducing redundancy and making it more organized.

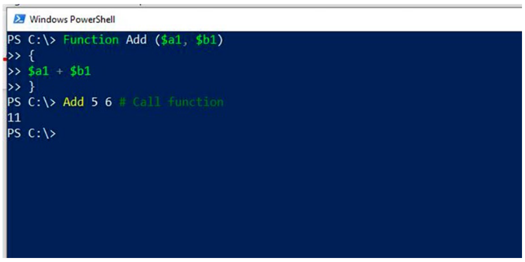
Functions are designed to encapsulate code into manageable units. By using functions, you avoid writing repetitive and lengthy code, making your scripts cleaner and easier to maintain.

In Listing 1-11, you create an Add function to add two numbers. The result is shown in Figure 1-19.

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

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

Add 5 6 # Call function.



Windows PowerShell

```
PS C:\> Function Add ($a1, $b1)
>> {
>> $a1 + $b1
>> }
PS C:\> Add 5 6 # Call function
11
PS C:\>
```

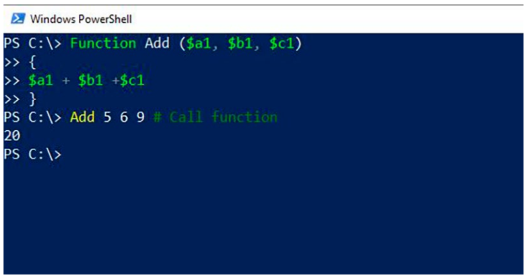
Figure 1-19. *Showing an Add function of two numbers*

Similarly, you can create this for three or more numbers. See Listing 1-12 and Figure 1-20.

Listing 1-12. Example Code Showing an Add Function of Three Numbers

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

Add 5 6 9 # Call function.



```
Windows PowerShell
PS C:\> Function Add ($a1, $b1, $c1)
>> {
>> $a1 + $b1 +$c1
>> }
PS C:\> Add 5 6 9 # Call function
20
PS C:\>
```

Figure 1-20. *Showing an Add function of three numbers*

Summary

In this chapter, you've explored fundamental concepts in PowerShell, including the following:

- **Variables:** How to store and manipulate data
- **Arrays:** Organizing multiple values into a single collection
- **If/Else Switch Statements:** Making decisions based on conditions
- **Loops:** Repeating actions until a condition is met

These foundational elements are crucial for crafting robust and effective scripts in any production environment. As you progress to the next chapters, you'll build on these basics to develop more advanced scripting skills and techniques.

CHAPTER 2

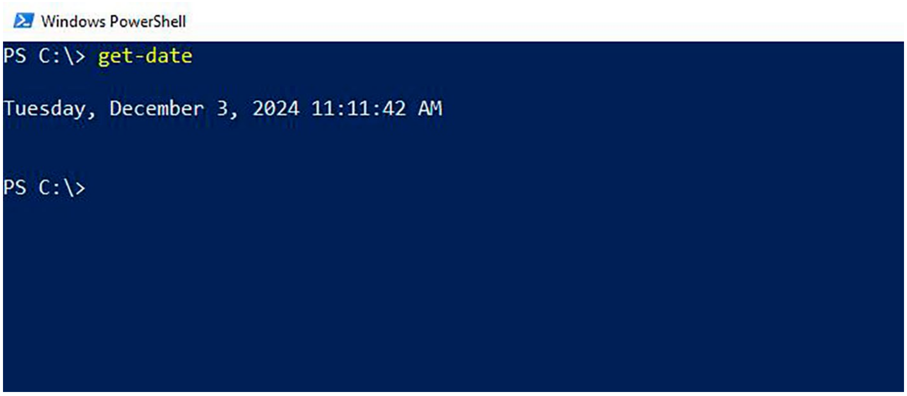
Date and Logs

Creating effective scripts often requires incorporating timestamps to track various operations. For example, you might need to create a time-stamped log file to record activities or insert time-stamped entries directly within the script. Understanding how to work with date and time cmdlets in PowerShell is crucial for achieving this.

In this section, I'll introduce a handy cheat function that you can use in your scripts to automatically generate time-stamped logs and entries. This function will simplify your workflow, ensuring that your logs are both accurate and easy to read.

Before diving into the **Write-Log** function (your first real cheat code!), let's explore some basic date and time operations. These examples will give you a solid foundation and help you see the practical application of date and time cmdlets in action.

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

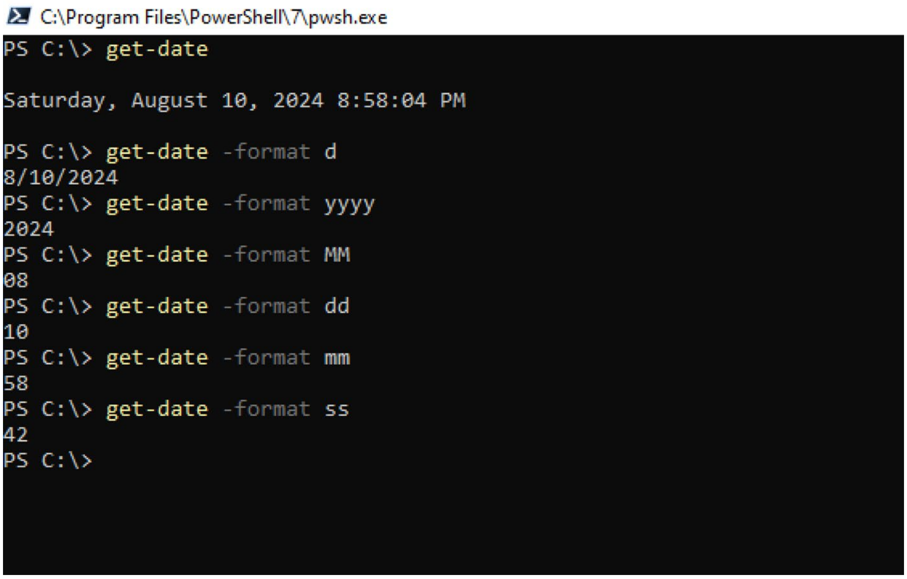


```
PS C:\> get-date

Tuesday, December 3, 2024 11:11:42 AM


PS C:\>
```

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
```



```
PS C:\> get-date

Saturday, August 10, 2024 8:58:04 PM

PS C:\> get-date -format d
8/10/2024
PS C:\> get-date -format yyyy
2024
PS C:\> get-date -format MM
08
PS C:\> get-date -format dd
10
PS C:\> get-date -format mm
58
PS C:\> get-date -format ss
42
PS C:\>
```

Figure 2-2. *Showing date formatting*

Common format specifiers include comma:

yyyy for the year

MM for the month

dd for the day

HH for the hour (24-hour clock)

mm for minutes

ss for seconds

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

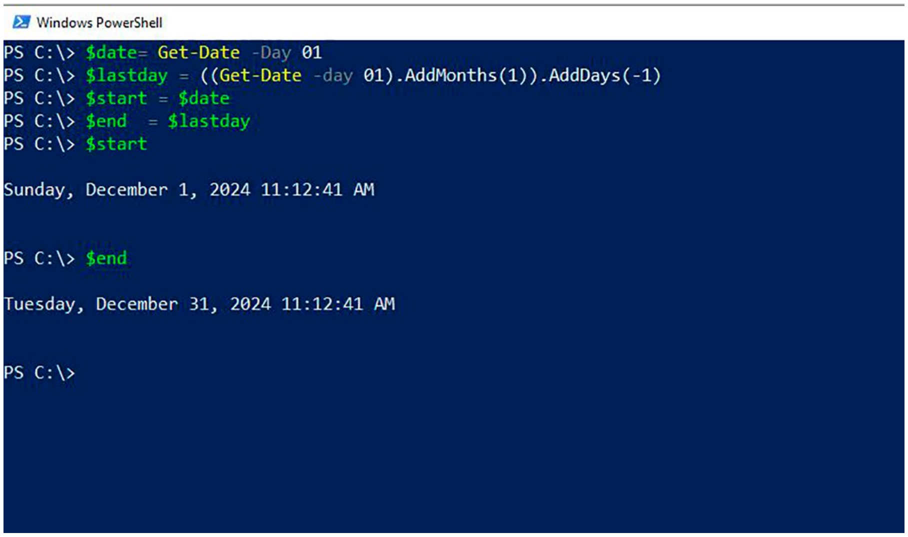
In the previous section, you saw how the Get-Date cmdlet provides the current date and time. This cmdlet is highly versatile, allowing you to manipulate date and time data to suit the specific needs of your scripting solution.

In this section, I'll briefly demonstrate how to perform essential operations, such as determining the first and last day of the month or generating a midnight timestamp. These operations are common in automation tasks where date boundaries are significant.

To get the first and last day of the month, use the code shown in Listing 2-2. This example illustrates how to capture the start and end dates of the current month. You can refer to Figure 2-3 for the resulting output.

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
```

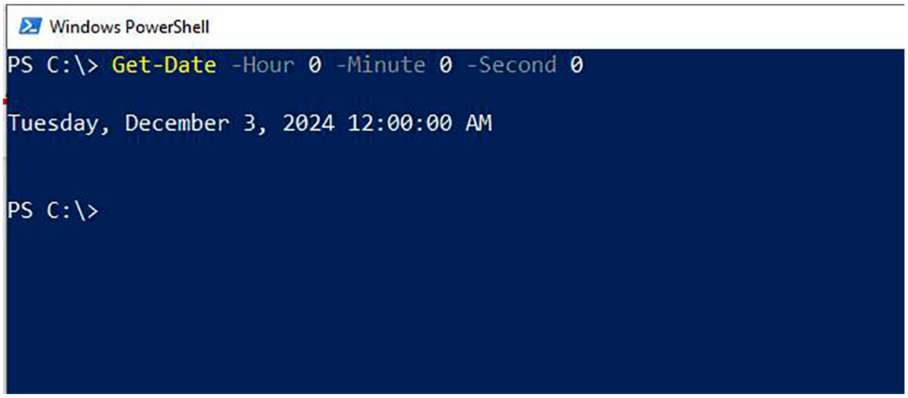


```
PS C:\> $date= Get-Date -Day 01
PS C:\> $lastday = ((Get-Date -day 01).AddMonths(1)).AddDays(-1)
PS C:\> $start = $date
PS C:\> $end  = $lastday
PS C:\> $start

Sunday, December 1, 2024 11:12:41 AM


PS C:\> $end

Tuesday, December 31, 2024 11:12:41 AM


PS C:\>
```

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
```



```
PS C:\> Get-Date -Hour 0 -Minute 0 -Second 0

Tuesday, December 3, 2024 12:00:00 AM


PS C:\>
```

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

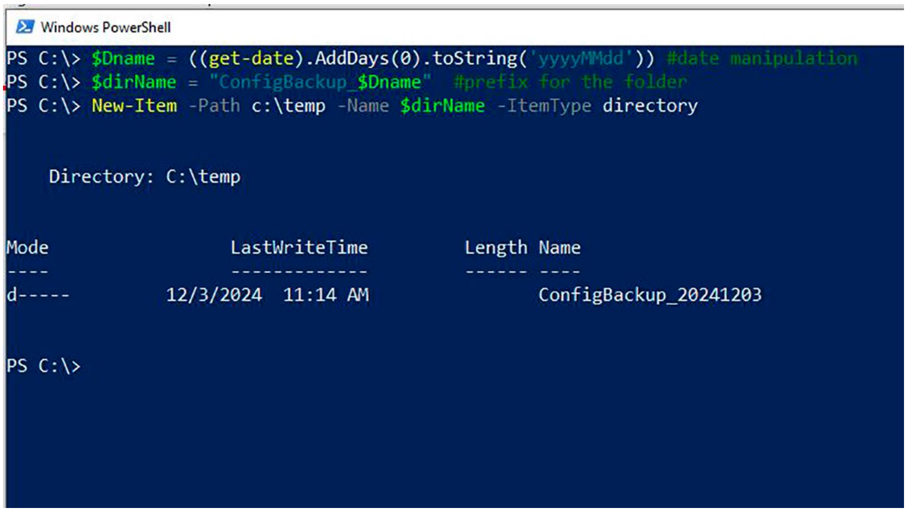
Creating Folders Based on a Date

In real-world scenarios, you may need to organize files by creating folders based on the current date. For instance, you might want to make daily backups of a SharePoint configuration, storing each backup in a uniquely named folder that reflects the date. This practice not only helps with organization but also ensures that you can easily locate backups from specific days.

Listing 2-3 demonstrates how you can achieve this by leveraging PowerShell's date-handling capabilities to create a folder named with the current date. Figure 2-5 illustrates the result of running this code.

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
```



```
Windows PowerShell
PS C:\> $Dname = ((get-date).AddDays(0).toString('yyyyMMdd')) #date manipulation
PS C:\> $dirName = "ConfigBackup_$Dname"  #prefix for the folder
PS C:\> New-Item -Path c:\temp -Name $dirName -ItemType directory


    Directory: C:\temp


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          12/3/2024  11:14 AM                ConfigBackup_20241203


PS C:\>
```

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. Toward 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 the following:

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

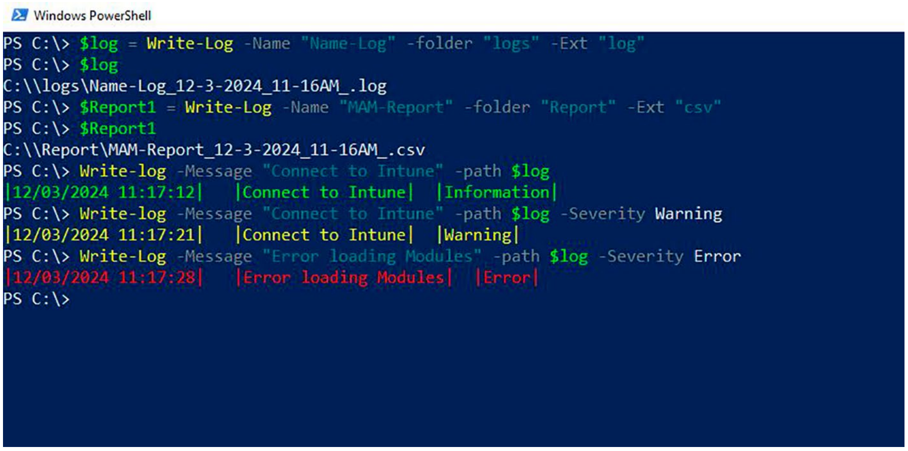
To write a warning to a log file, you can use the following:

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

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

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

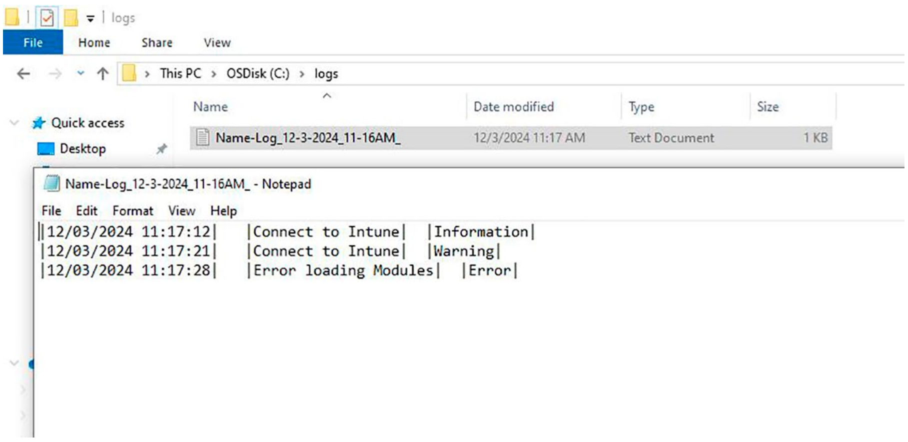
Figure 2-6 shows the `Write-Log` operation in the PowerShell console.



```
PS C:\> $log = Write-Log -Name "Name-Log" -folder "logs" -Ext "log"
PS C:\> $log
C:\\logs\Name-Log_12-3-2024_11-16AM_.log
PS C:\> $Report1 = Write-Log -Name "MAM-Report" -folder "Report" -Ext "csv"
PS C:\> $Report1
C:\\Report\MAM-Report_12-3-2024_11-16AM_.csv
PS C:\> Write-log -Message "Connect to Intune" -path $log
|12/03/2024 11:17:12|    |Connect to Intune|   |Information|
PS C:\> Write-log -Message "Connect to Intune" -path $log -Severity Warning
|12/03/2024 11:17:21|    |Connect to Intune|   |Warning|
PS C:\> Write-Log -Message "Error loading Modules" -path $log -Severity Error
|12/03/2024 11:17:28|    |Error loading Modules|   |Error|
PS C:\>
```

Figure 2-6. Write-Log operation

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


logs
File
Home
Share
View
This PC > OSDisk (C:) > logs
Quick access
Desktop
Name
Date modified
Type
Size
Name-Log_12-3-2024_11-16AM_
12/3/2024 11:17 AM
Text Document
1 KB
Name-Log_12-3-2024_11-16AM_ - Notepad
File Edit Format View Help

```
|12/03/2024 11:17:12|    |Connect to Intune|   |Information|
|12/03/2024 11:17:21|    |Connect to Intune|   |Warning|
|12/03/2024 11:17:28|    |Error loading Modules|   |Error|
```

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 specify 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:

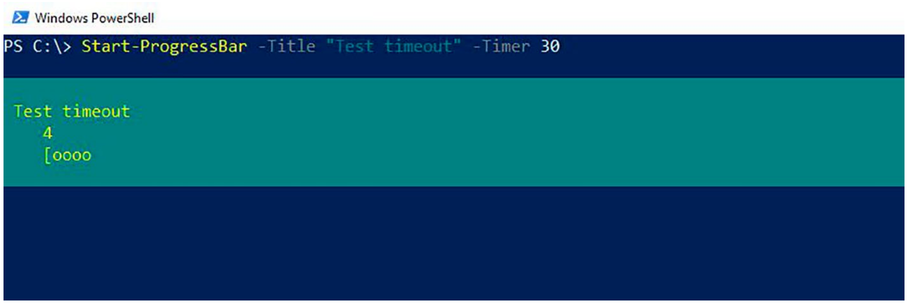
```
Set-Recyclelogs -ComputerName testmachine -DriveName
c  -folderpath data\logs -limit 10
```

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

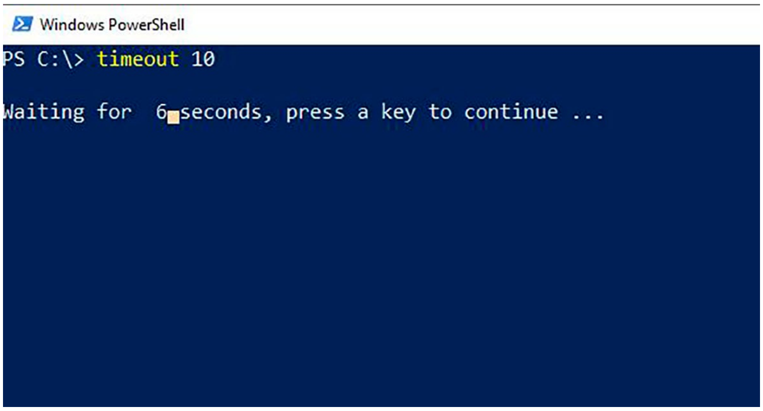


```
PS C:\> Start-ProgressBar -Title "Test timeout" -Timer 30

Test timeout
   4
   [oooo
```

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
```



```
PS C:\> timeout 10

Waiting for  6 seconds, press a key to continue ...
```

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

To streamline and automate various tasks within your PowerShell environment, I have developed a custom module named vsadmin. This module contains a comprehensive set of functions tailored to meet the needs of IT professionals looking for efficient solutions.

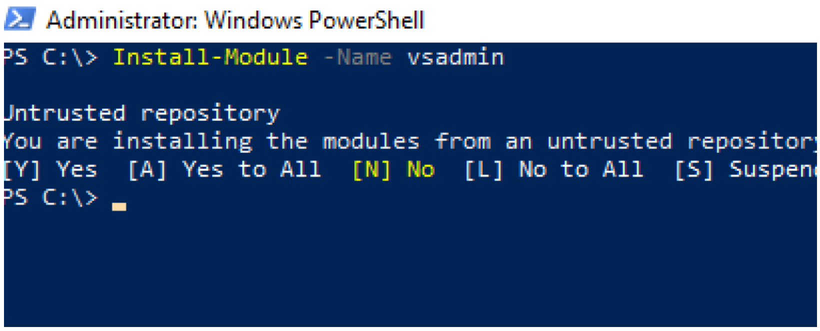
You can easily install the vsadmin module from the PowerShell Gallery by executing the following command:

`https://www.powershellgallery.com/packages/vsadmin`

Install-Module -Name vsadmin (Figure 2-10)

In the event that you have already installed the vsadmin module, you might want to ensure that you are using the latest version. To upgrade the module to the latest version, use the -Force parameter:

Install-Module -Name vsadmin -Force



```
PS C:\> Install-Module -Name vsadmin

Untrusted repository
You are installing the modules from an untrusted repositor
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspen
PS C:\> _
```

Figure 2-10. *Install-Module vsadmin*

While this chapter introduces the installation and upgrade process of vsadmin, I have written a dedicated chapter that thoroughly explains all the functionalities and usage of the vsadmin module. You can find this chapter later in the book, where I have provided examples and scenarios to help you make the most of this powerful tool.

Summary

In this chapter, you explored the powerful capabilities of PowerShell's date and log cmdlets, learning how to manipulate and format dates to generate time-stamped folders and files. These techniques are invaluable in a variety of real-world scenarios, such as automatically creating daily log files or organizing data into date-specific folders.

By mastering these skills, you can enhance the automation and organization of your scripts, ensuring that your files are systematically named and easily accessible. Whether you're creating backup directories or logging operational data, incorporating date and timestamps into your file and folder names will streamline your workflows and improve the efficiency of your scripts.

CHAPTER 3

Input to Your Scripts

In the real world of system administration, it's common to encounter scenarios where your scripts need to process and act on various types of input data. For instance, you might need to read a text file containing a list of users to add them to a specific Active Directory group. Alternatively, you might have a CSV file that includes user attributes like phone numbers, job titles, and departments, which need to be updated in Active Directory.

These tasks are not just routine—they are crucial for maintaining the accuracy and efficiency of your IT infrastructure. The ability to automate these processes can save you significant time and reduce the risk of human error.

In this chapter, we'll explore the different methods available in PowerShell for feeding your scripts with various types of inputs. Whether you're dealing with simple text files, structured CSV files, or more complex data sources, understanding these techniques will empower you to create more dynamic and flexible scripts.

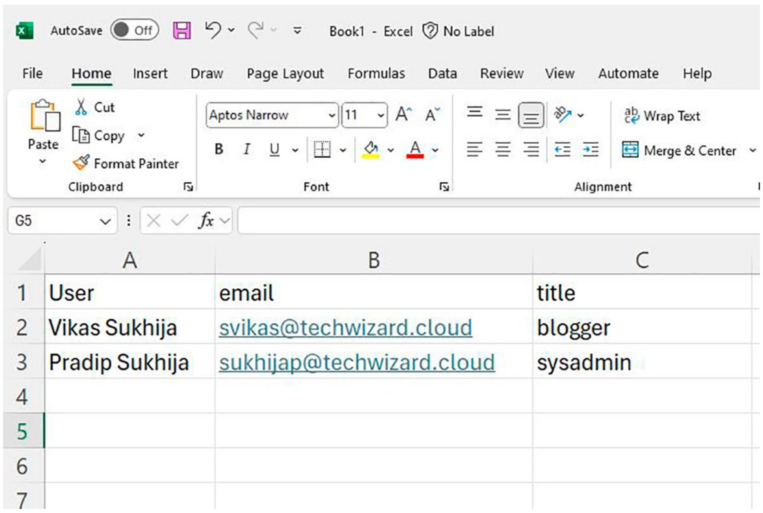
We will delve into practical examples, guiding you through the process of reading, processing, and utilizing input data effectively. By the end of this chapter, you'll be equipped with the knowledge to handle a wide range of input scenarios, making your automation scripts more powerful and adaptable to the demands of your system administration tasks.

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_3

Import-CSV

One of the most common methods for providing a script with structured input data is by using the Import-CSV cmdlet. This cmdlet is particularly useful when dealing with bulk operations, as it allows you to easily read data from a CSV file and process it within a loop.

To illustrate this, let's start by creating a small CSV file that we'll name samplecsv.csv. This file will contain sample data in the format shown in Figure 3-1. Once we have the file, we'll use PowerShell to import the data and print its contents to the console by utilizing code in Listing 3-1



	A	B	C
1	User	email	title
2	Vikas Sukhija	svikas@techwizard.cloud	blogger
3	Pradip Sukhija	sukhijap@techwizard.cloud	sysadmin
4			
5			
6			
7			

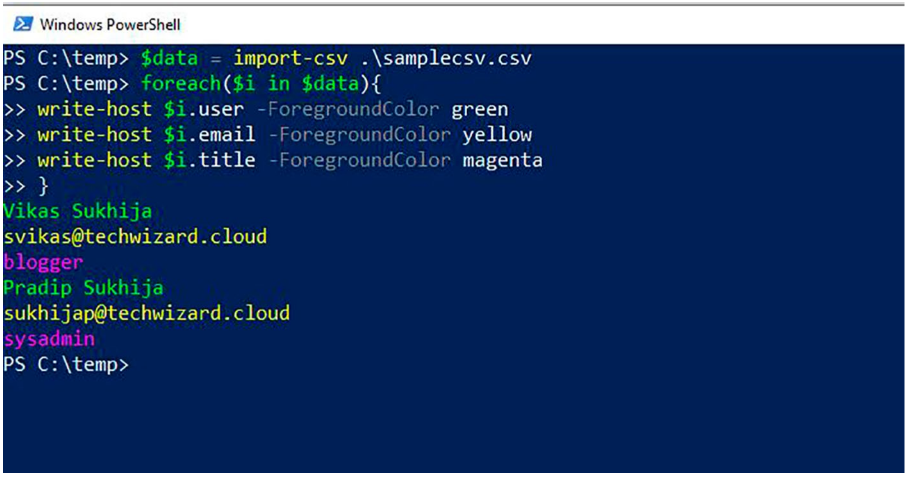
Figure 3-1. *Example CSV file*

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're working within the same directory as your CSV file, you can simplify the command by using a relative path. Instead of specifying the full path to the file, you can use a dot (.) to represent the current directory, as shown below:

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



Windows PowerShell

```
PS C:\temp> $data = import-csv .\samplecsv.csv
PS C:\temp> foreach($i in $data){
>> write-host $i.user -ForegroundColor green
>> write-host $i.email -ForegroundColor yellow
>> write-host $i.title -ForegroundColor magenta
>> }
Vikas Sukhija
svikas@techwizard.cloud
blogger
Pradip Sukhija
sukhijap@techwizard.cloud
sysadmin
PS C:\temp>
```

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

There are various practical examples of importing CSV files in PowerShell. For instance, you might need to import data from a CSV file to update Active Directory user attributes or transfer information to a third-party system.

A common scenario in organizations is receiving a daily CSV feed containing user attributes that need to be updated in Active Directory. These attributes could include details such as state, city, country, job codes, addresses, phone numbers, and more.

By combining the Import-Csv cmdlet with Active Directory or other module cmdlets, you can efficiently pipe the imported data to update user attributes, as illustrated below in Listing 3-2.

Listing 3-2. Practical Example of Importing CSV Feed into Active Directory

```
Csv file fortmat:
SamAccountName,City,State,Country,JobTitle
jdoe,New York,NY,USA,Manager
asmith,Los Angeles,CA,USA,Developer
bjones,Chicago,IL,USA,Analyst

# Import the Active Directory module
Import-Module ActiveDirectory

# Import the CSV file
$users = Import-Csv ".\UserData.csv"

# Iterate through each user in the CSV and update the
attributes in AD
foreach ($user in $users) {
    # Update the user attributes in Active Directory
    Set-ADUser -Identity $user.SamAccountName `
               -City $user.City `
               -State $user.State `
               -Country $user.Country `
               -Title $user.Title
    # Output the updated user details
    Write-Host "Updated attributes for user: $($user.
    SamAccountName)"
}
```

Instead of using write host, you can combine the Write-Log function that we have learned in the last chapter and create log.

Here is the updated version of the script in Listing 3-3.

Listing 3-3. Update the Script in Listing 3-2 with Log Function from Previous Chapter

```
#Create log variable (either import write-log #fundtion or
install vsadmin module)
$log = Write-Log -Name "ADattributes-Log" -folder "logs"
-Ext "log"
# Import the Active Directory module
Import-Module ActiveDirectory

# Import the CSV file
$users = Import-Csv ".\UserData.csv"

# Iterate through each user in the CSV and update the
attributes in AD
foreach ($user in $users) {
    # Update the user attributes in Active Directory
    Set-ADUser -Identity $user.SamAccountName `
               -City $user.City `
               -State $user.State `
               -Country $user.Country `
               -Title $user.Title
    # Output the updated user details
    Write-Log -message "Updated attributes for user: $($user.
    SamAccountName)" -path $log
}
```

With this enhanced version, you'll have a persistent log of all activities performed by the script. This is crucial for auditing, troubleshooting, and maintaining a record of changes made to user attributes in Active Directory.

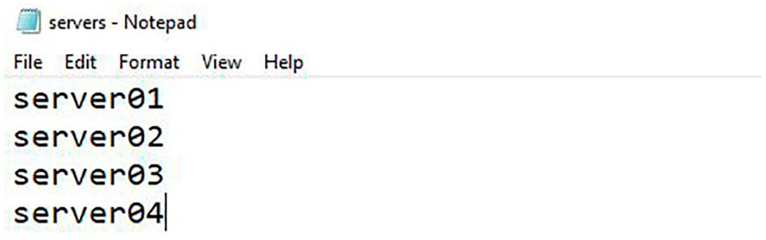
As you progress through the book, you can gradually enhance this script by incorporating advanced features such as error handling, alerting, and more. These additions will not only make the script more robust but also provide readers with a comprehensive understanding of building powerful and reliable PowerShell scripts.

Importing from a Text File

In some scenarios, you may receive data in a simple text file, such as a server list or a user list, with each entry on a new line. You might need to perform specific operations on this data, such as iterating through the list and executing commands for each entry.

For example, let's say you have a file named servers.txt that contains a list of server names, one per line.

Figure 3-3 illustrates the content of this file.



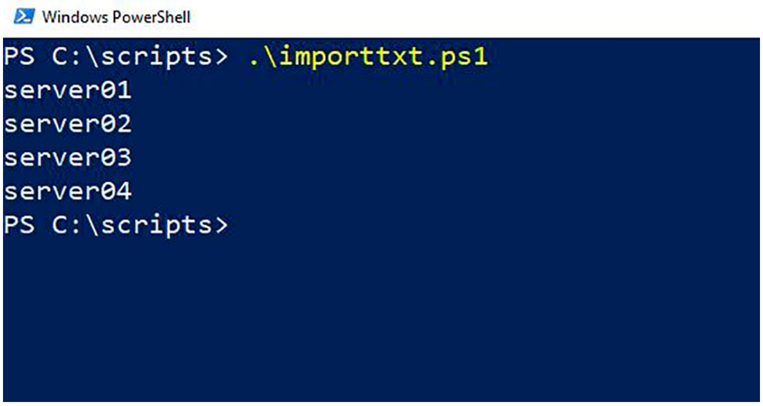
```
server01
server02
server03
server04
```

Figure 3-3. *Example text file contents*

Listing 3-4. Code for Reading from a Text File

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

To read the contents of this file and print each server name to the screen, you can use the Get-Content cmdlet. This cmdlet reads the contents of the file and stores it in a variable. You can then use a foreach-object loop to iterate through each line of the file. The following script demonstrates this process (Figure 3-4).



```
PS C:\scripts> .\importtxt.ps1
server01
server02
server03
server04
PS C:\scripts>
```

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

Save the Listing 3-4 as importtxt.ps1 and then run in powershell console.

In practical scenarios where you need to perform actions on a list of servers, such as sending shutdown or restart commands, you can use a similar approach to what we've discussed. By leveraging PowerShell's capabilities, you can automate these tasks efficiently.

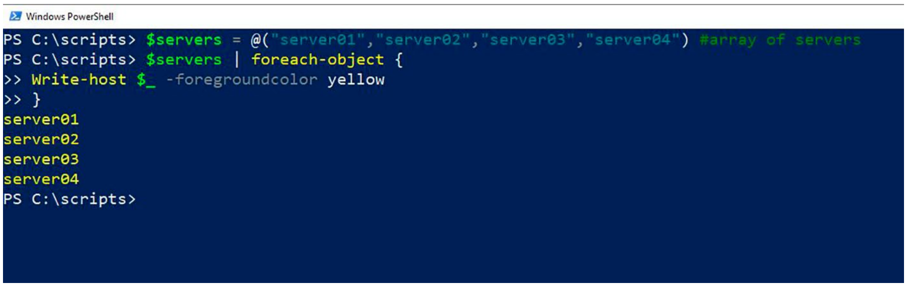
Input from an Array

You can perform similar operations with arrays as you did with text files. For instance, if you have an array of server names and want to print each one to the screen, you can use the ForEach-Object cmdlet. See Listing 3-5.

Listing 3-5. 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 as shown in Figure 3-5.



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

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

Each server name is printed in yellow text, demonstrating how you can handle and display array data in PowerShell.

This approach is straightforward and works well for scenarios where data is already available in an array, allowing you to quickly iterate and perform operations on each item.

User Input with Prompts

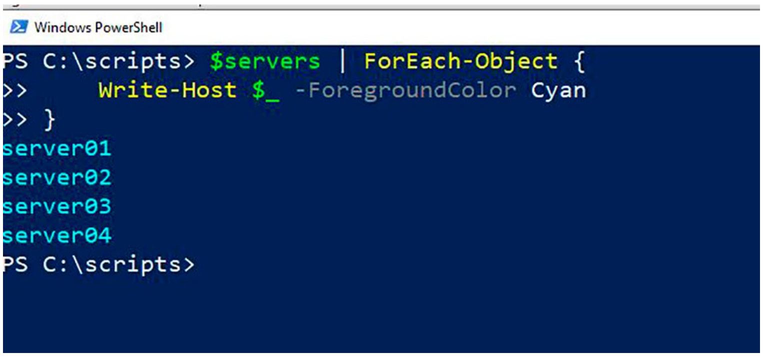
In PowerShell, you can use the Read-Host cmdlet to prompt the user for input. This method allows you to interactively gather information from the user and use it within your script.

Listing 3-6. Code for Prompting User Input and Printing It

```
# Prompt the user to enter server names, separated by commas
$input = Read-Host "Enter server names, separated by commas"

# Split the input string into an array of server names
$servers = $input -split ",\s*"
```

```
# Iterate through each server in the array and print it to
the screen
$servers | ForEach-Object {
    Write-Host $_ -ForegroundColor Cyan
}
```

Save Listing 3-6 as inputfromprompt.ps1 and run in powershell; output will be as shown in Figure 3-6.



```
PS C:\scripts> $servers | ForEach-Object {
>>     Write-Host $_ -ForegroundColor Cyan
>> }
server01
server02
server03
server04
PS C:\scripts>
```

Figure 3-6. *Showing the printing of user input*

In a similar fashion, you might need to obtain a password for running a process securely. To ensure that the password is not exposed in plain text, you can use the **-AsSecureString** parameter with Read-Host to collect the password as a secure string.

```
$securePassword = Read-Host "Enter your password"
-AsSecureString
```

We will cover more on interactive inputs in our next chapter which is dedicated to it.

Summary

In this chapter, we explored various methods for providing input to PowerShell scripts, focusing on practical and commonly used techniques in system administration. Specifically, you learned how to

Use Text Files: Read and process data from a text file using Get-Content, which allows you to handle simple lists of items, such as server names.

Work with CSV Files: Import and manipulate data from CSV files with Import-Csv, enabling you to update attributes or perform actions based on structured data, such as user details for Active Directory.

Use of Arrays as Input: Manage arrays to store and iterate over data directly within scripts, making it easy to process a predefined list of items.

Prompt for User Input: Securely gather sensitive information, such as passwords, using Read-Host with the -AsSecureString parameter, ensuring that credentials are handled securely.

These methods provide a strong foundation for scripting and automation tasks. While there are more advanced techniques for feeding scripts with input, the approaches covered in this chapter are essential for everyday system administration and offer practical solutions for a variety of common scenarios.

CHAPTER 4

Interactive Input

In this chapter, we'll delve into how to add interactive input capabilities to your PowerShell scripts. Interactive input allows your script to engage with the user, prompting them for necessary details during execution. This is particularly useful for scenarios where dynamic or user-specific information is required.

Read-Host

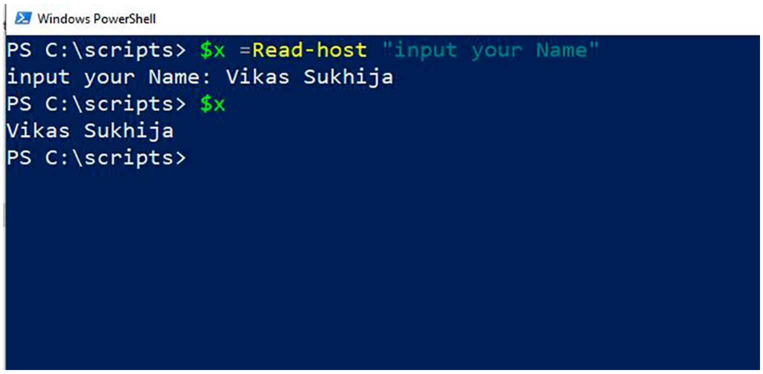
The most fundamental method for interactive input in PowerShell is Read-Host. This cmdlet prompts the user to enter information and captures the input as a string.

To illustrate how Read-Host works, consider the following example:

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

In this example, Read-Host displays the prompt "Input your Name" on the screen. The user types their name, and the entered value is assigned to the variable $x. You can then use this variable for further processing in your script (Figure 4-1).

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_4

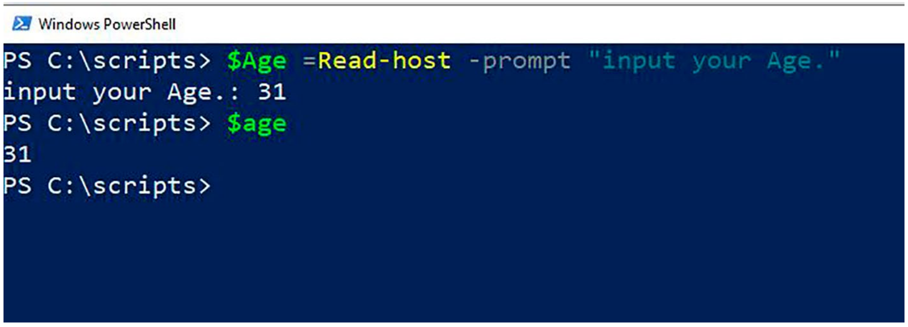


```
PS C:\scripts> $x =Read-host "input your Name"
input your Name: Vikas Sukhija
PS C:\scripts> $x
Vikas Sukhija
PS C:\scripts>
```

Figure 4-1. *Read-Host operation*

You can explicitly use -prompt parameter as well while writing Read-Host command as shown in Figure 4-2. It helps make your script more readable and the prompt message more explicit.

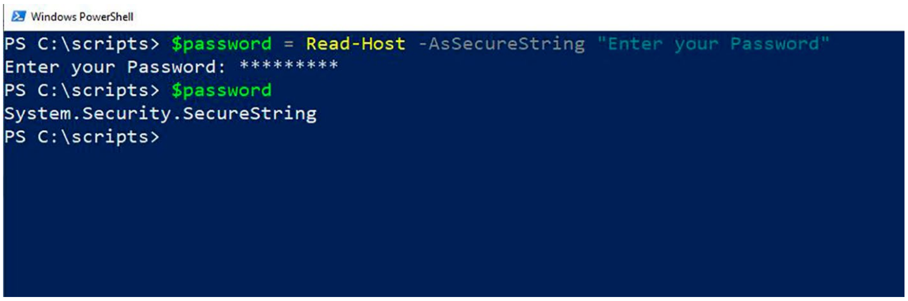
```
$Age =Read-host -prompt "input your Age."
```



```
PS C:\scripts> $Age =Read-host -prompt "input your Age."
input your Age.: 31
PS C:\scripts> $age
31
PS C:\scripts>
```

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

As mentioned in the previous chapter, you can use the -AsSecureString parameter with Read-Host to securely capture sensitive information, such as passwords (Figure 4-3). This ensures that the input is encrypted and not exposed as plain text, enhancing the security of your script.



```
PS C:\scripts> $password = Read-Host -AsSecureString "Enter your Password"
Enter your Password: *********
PS C:\scripts> $password
System.Security.SecureString
PS C:\scripts>
```

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

In this example, the password entered by the user is stored in the $Password variable as a secure string. This secure string can then be used in your script for various purposes, such as authentication to services like Office 365.

Using Read-Host, you can prompt the user or administrator to input the path to CSV file or text file that you want to process further.

As covered in the previous chapter, you learned how to read data from files, such as CSV or text files, and process it by piping the data to other commands. Instead of hardcoding file paths into your script, you can enhance its flexibility by prompting users to provide the file path interactively.

Input Parameters

In PowerShell, commands, functions, and scripts often rely on parameters as shared in Listing 4-1 example, which allow users to enter values or select options to customize the script's behavior. In this section, we'll briefly touch on basic parameterization, providing you with the foundational knowledge to use parameters in your scripts. (Note that advanced parameters are outside the scope of this book.)

Listing 4-1. An Example of How to Define and Use Parameters in a PowerShell Script

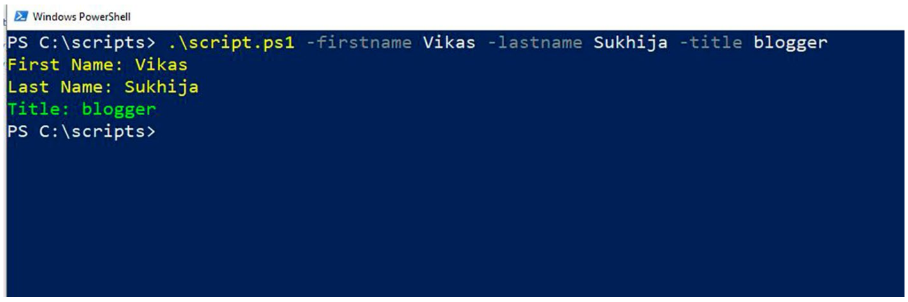
```
Param(
  [string]$firstname,
  [string]$lastname,
  [string]$title
)
Write-host "First Name: $firstname" -ForegroundColor Yellow
Write-host "Last Name: $lastname" -ForegroundColor Yellow
Write-host "Title: $Title" -ForegroundColor green
```

In this example:

- **Param Block**: The Param block at the beginning of the script defines three parameters: $firstname, $lastname, and $title, all of which are of the type [string].
- **Parameter Usage:** The script uses Write-Host to output the values of these parameters, with each output line in a specified color.

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

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



```
PS C:\scripts> .\script.ps1 -firstname Vikas -lastname Sukhija -title blogger
First Name: Vikas
Last Name: Sukhija
Title: blogger
PS C:\scripts>
```

Figure 4-4. *Script execution with parameters*

Tip Make sure you define the parameters at the beginning of your script; otherwise, the script will not work. Placing the Param block at the top not only organizes your code but also ensures that PowerShell correctly recognizes and processes the parameters when the script runs.

GUI Button

If you want to create a more interactive and visually appealing way to gather input from users, you can use a graphical user interface (GUI) button. While not as commonly used as the Read-Host command, a GUI button can be a “fancy” way to get input from the user, making your script more engaging.

Here’s a cheat code—a function—that you can use to create a GUI button in your PowerShell script. This function, shown in Listing 4-2, allows users to input data through a Windows form, which can then be used to perform other desired operations in your script.

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

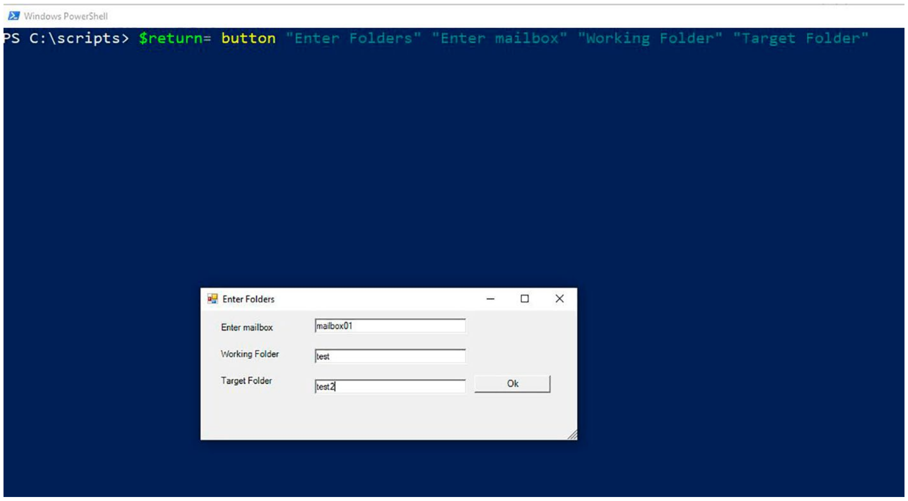
```
function button ($title,$mailbx, $WF, $TF)
{

###################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 or parameters for your GUI input according to your requirements. Figure 4-5 illustrates the Windows form with the input textbox and button:

By integrating this function into your script, you can enhance user interaction by providing a simple and intuitive way to input data through a GUI, which can be especially useful for those who are less comfortable with command-line inputs.

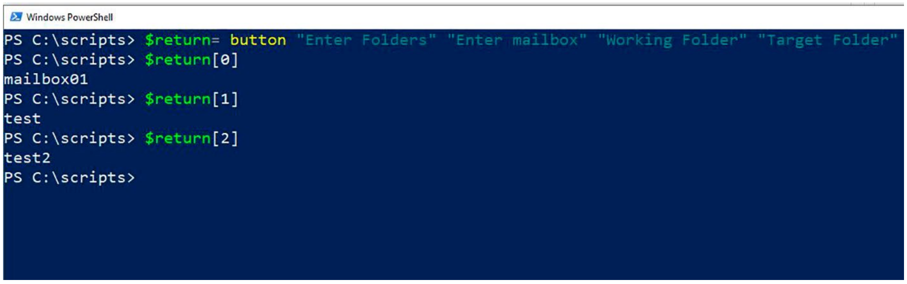


```
PS C:\scripts> $return= button "Enter Folders" "Enter mailbox" "Working Folder" "Target Folder"
```

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
```

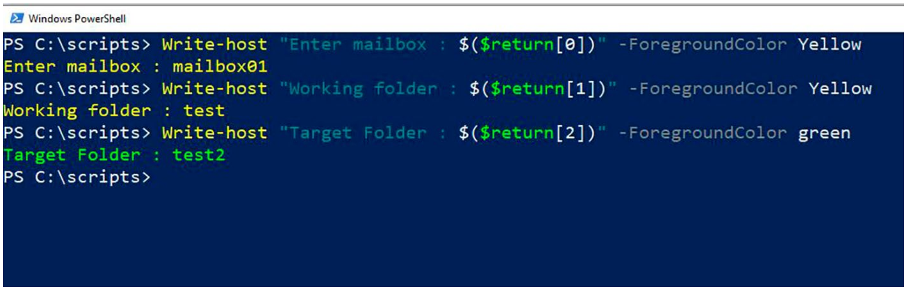


```
PS C:\scripts> $return= button "Enter Folders" "Enter mailbox" "Working Folder" "Target Folder"
PS C:\scripts> $return[0]
mailbox01
PS C:\scripts> $return[1]
test
PS C:\scripts> $return[2]
test2
PS C:\scripts>
```

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
```



```
PS C:\scripts> Write-host "Enter mailbox : $($return[0])" -ForegroundColor Yellow
Enter mailbox : mailbox01
PS C:\scripts> Write-host "Working folder : $($return[1])" -ForegroundColor Yellow
Working folder : test
PS C:\scripts> Write-host "Target Folder : $($return[2])" -ForegroundColor green
Target Folder : test2
PS C:\scripts>
```

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

Prompt (Yes or No)

As a system administrator, there are many practical situations where you might need to prompt users for a simple Yes/No response. Whether you're asking for confirmation before proceeding with an action or offering users a choice, having a clear and user-friendly method for gathering these responses is essential.

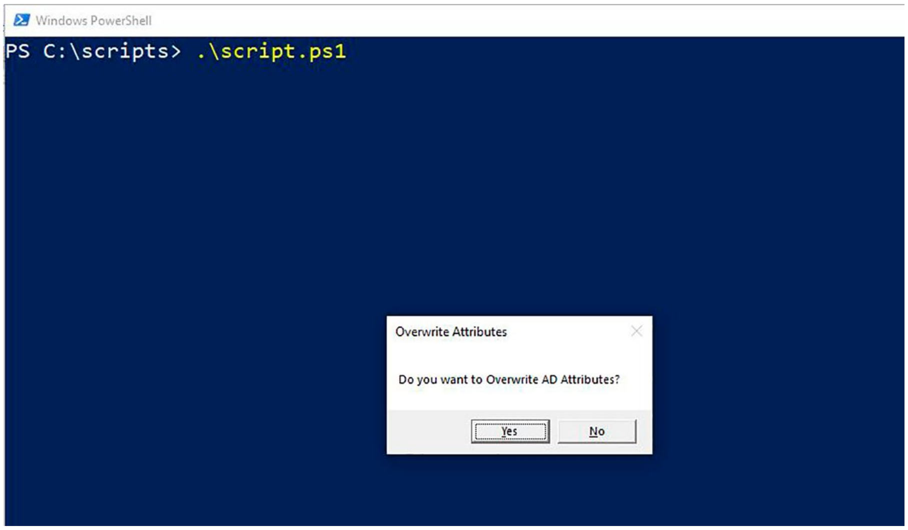
PowerShell provides a straightforward way to achieve this using a GUI prompt, which can be particularly useful when you want to ensure users clearly understand the choices available. The cheat code in Listing 4-3 demonstrates how to create a Yes/No prompt using PowerShell.

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.

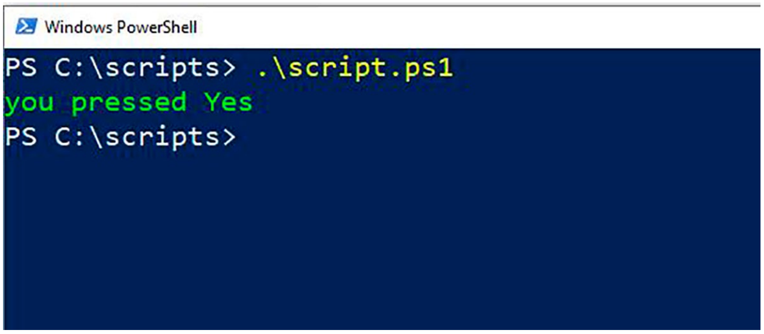


```
PS C:\scripts> .\script.ps1
```

Do you want to Overwrite AD Attributes?

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

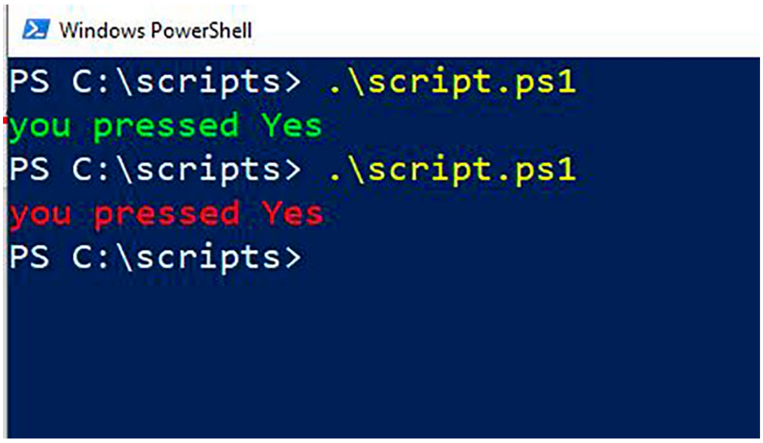
If you press Yes, you get the result shown in Figure 4-9.



```
PS C:\scripts> .\script.ps1
you pressed Yes
PS C:\scripts>
```

Figure 4-9. *Showing the Yes operation*

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



Windows PowerShell

```
PS C:\scripts> .\script.ps1
you pressed Yes
PS C:\scripts> .\script.ps1
you pressed Yes
PS C:\scripts>
```

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.

This method not only ensures that users are making informed decisions but also adds a layer of confirmation to your scripts, reducing the risk of accidental actions. Whether you're deploying updates, deleting files, or executing critical tasks, a simple Yes/No prompt can be an effective tool in your PowerShell scripting arsenal.

Summary

In this chapter, you learned about interactive inputs, an essential strategy for making scripts more user-friendly, especially when they are intended for use by end users. By incorporating interactive prompts, you can create scripts that guide users through a series of questions, allowing them to provide necessary input without needing to modify the script directly.

This approach enhances the usability of your scripts, as end users can simply run the script and respond to the prompts that appear. Whether it's entering a password securely, selecting a file path, or confirming an action with a Yes/No response, interactive inputs make the script execution process more intuitive and efficient. As a result, users can perform meaningful tasks with minimal effort, while the scripter ensures that the script operates as intended with the correct input.

CHAPTER 5

Modules

Microsoft and various third-party vendors have developed PowerShell snap-ins (which are now mostly outdated) or modules for their respective products. To leverage the PowerShell cmdlets designed for these technologies, you would need to either add the snap-ins or import the modules into your scripts.

Snap-ins are considered a legacy approach, as the PowerShell ecosystem has largely shifted toward using **modules**. Modules are more versatile and easier to manage, effectively serving as “batteries” that power your scripts. Each module is essentially a package that can contain cmdlets, providers, functions, aliases, and other resources necessary to automate or interact with a product.

Although snap-ins are no longer common, it’s still valuable to briefly touch on them to understand their role and the products that historically relied on them. Many older technologies, particularly those from earlier versions of PowerShell, used snap-ins to extend PowerShell’s functionality before modules became the standard.

PowerShell Snap-Ins

A classic example of PowerShell snap-ins is **Microsoft Exchange Server**, specifically versions 2007 and 2010, which both utilized snap-ins for their PowerShell integration.

© Vikas Sukhija 2025
V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_5

To add an Exchange snap-in to your script, you could use the following code (see Listings 5-1 and 5-2 for examples of Exchange 2007 and 2010 snap-ins, respectively).

Since these versions of Exchange have reached their official end-of-support status, their usage is rare today, primarily limited to organizations that have not yet upgraded to newer versions of Exchange Server, which use PowerShell modules instead.

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

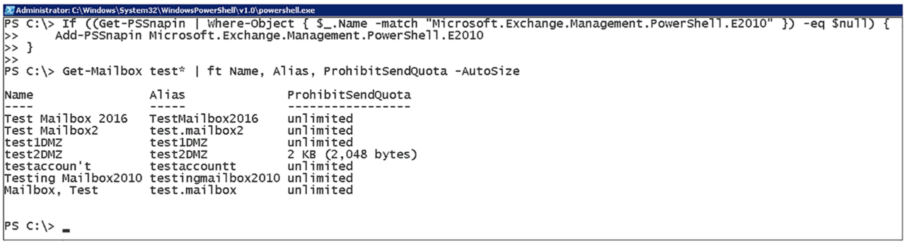
Listing 5-1. Code to Add the Exchange 2007 Management Shell

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

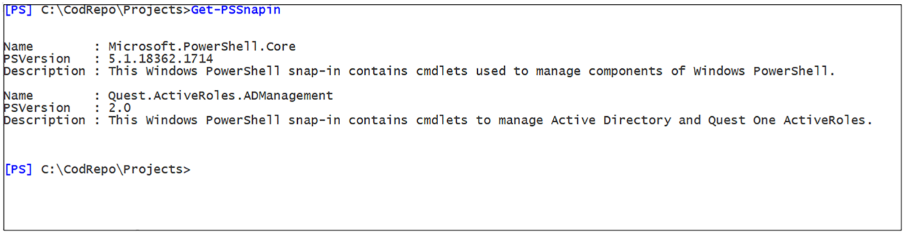


```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\> If ((Get-PSSnapin | Where-Object { $_.Name -match "Microsoft.Exchange.Management.PowerShell.E2010" }) -eq $null) {
>>     Add-PSSnapin Microsoft.Exchange.Management.PowerShell.E2010
>> }
>>
PS C:\> Get-Mailbox test* | ft Name, Alias, ProhibitSendQuota -AutoSize

Name                 Alias                ProhibitSendQuota
----                 -----                -----------------
Test Mailbox 2016    TestMailbox2016      unlimited
Test Mailbox2        test.mailbox2        unlimited
test1DMZ             test1DMZ             unlimited
test2DMZ             test2DMZ             2 KB (2,048 bytes)
testaccoun't         testaccountt         unlimited
Testing Mailbox2010  testingmailbox2010   unlimited
Mailbox, Test        test.mailbox         unlimited

PS C:\> _
```

Figure 5-1. *Exchange Management Shell*

You can also use the Get-PSSnapin cmdlet within a PowerShell session to check which snap-ins are currently available. This can be helpful when working with legacy systems to verify the snap-ins loaded into your environment.

For example, in Figure 5-2, I demonstrate using the Get-PSSnapin cmdlet within the Quest AD shell to take advantage of the Quest snap-in. Running Get-PSSnapin in the Quest Active Directory shell allows you to identify the specific snap-in name and confirm it's properly loaded for use in your scripts.



```
[PS] C:\CodRepo\Projects>Get-PSSnapin


Name        : Microsoft.PowerShell.Core
PSVersion   : 5.1.18362.1714
Description : This Windows PowerShell snap-in contains cmdlets used to manage components of Windows PowerShell.

Name        : Quest.ActiveRoles.ADManagement
PSVersion   : 2.0
Description : This Windows PowerShell snap-in contains cmdlets to manage Active Directory and Quest One ActiveRoles.


[PS] C:\CodRepo\Projects>
```

Figure 5-2. *Quest AD Management Shell*

Listing 5-3 shows how to add the snap-in 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 snap-in first checks to see if the snap-in already exists. If so, it does nothing. If not, it adds the required snap-in.

Modules

Now, let's shift our focus to **PowerShell modules**, as they are integral to your day-to-day work. According to Microsoft, *"a module is a package that contains PowerShell members, such as cmdlets, providers, functions, workflows, variables, and aliases. These members can be implemented in a PowerShell script, a compiled DLL, or a combination of both. Typically, these files are grouped together in a single directory."*

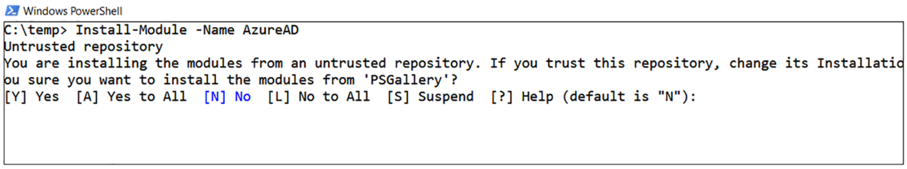
Modules are essential for interacting with a wide range of products and services, such as **Azure**, **Office 365**, **Exchange Online**, **Microsoft Teams, Sharepoint Online**, **Amazon Web Services**, and many other products. Instead of relying on legacy snap-ins, modern PowerShell heavily utilizes modules to provide access to the cmdlets required for automating and managing these technologies.

The **PowerShell Gallery** (available at `www.powershellgallery.com`) serves as the central repository for nearly all publicly available PowerShell modules. It is the de facto resource where you can find and install modules to expand PowerShell's capabilities, whether you're working with cloud services, on-premises systems, or third-party tools.

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

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

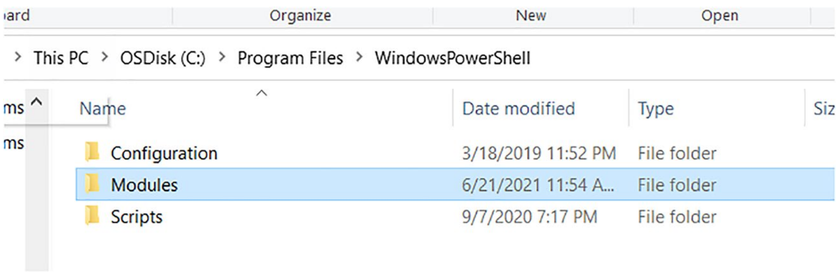
Enter yes (as shown in Figure 5-3) when you receive the prompt to install the module.



```
C:\temp> Install-Module -Name AzureAD
Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repository, change its Installatio
ou sure you want to install the modules from 'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"):
```

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.



Name	Date modified	Type	Siz
Configuration	3/18/2019 11:52 PM	File folder	
Modules	6/21/2021 11:54 A...	File folder	
Scripts	9/7/2020 7:17 PM	File folder	

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
```

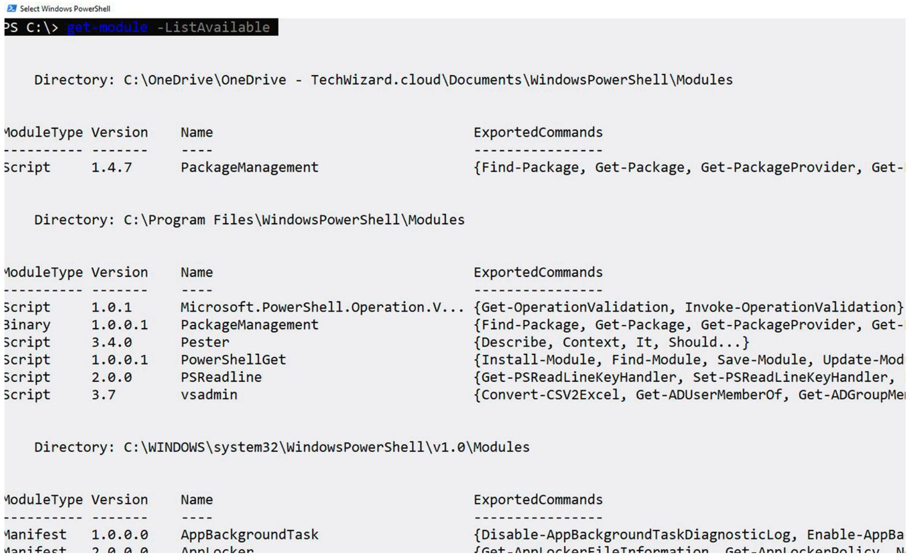

```
PS C:\> get-module -ListAvailable


    Directory: C:\OneDrive\OneDrive - TechWizard.cloud\Documents\WindowsPowerShell\Modules


ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     1.4.7      PackageManagement                   {Find-Package, Get-Package, Get-PackageProvider, Get-


    Directory: C:\Program Files\WindowsPowerShell\Modules


ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     1.0.1      Microsoft.PowerShell.Operation.V... {Get-OperationValidation, Invoke-OperationValidation}
Binary     1.0.0.1    PackageManagement                   {Find-Package, Get-Package, Get-PackageProvider, Get-
Script     3.4.0      Pester                              {Describe, Context, It, Should...}
Script     1.0.0.1    PowerShellGet                       {Install-Module, Find-Module, Save-Module, Update-Mod
Script     2.0.0      PSReadline                          {Get-PSReadLineKeyHandler, Set-PSReadLineKeyHandler,
Script     3.7        vsadmin                             {Convert-CSV2Excel, Get-ADUserMemberOf, Get-ADGroupMe


    Directory: C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules


ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Manifest   1.0.0.0    AppBackgroundTask                   {Disable-AppBackgroundTaskDiagnosticLog, Enable-AppBa
```

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

Cheat Module (vsadmin)

Since this book is all about providing cheat codes to help you create complex scripts, here's a powerful resource: a **cheat module** packed with functions designed to streamline complex operations in your scripts.

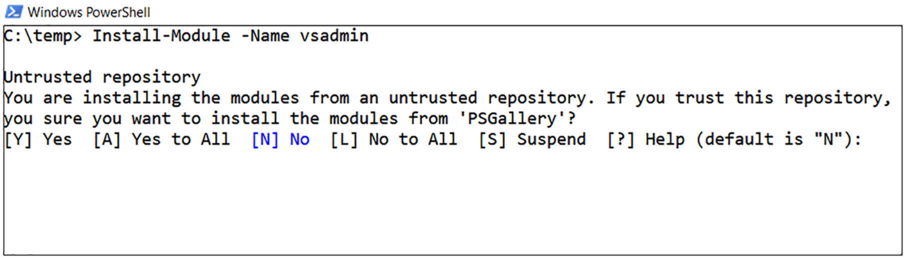
I have developed this module for the PowerShell community, aiming to simplify scripting by offering ready-made functions that can be adapted to a variety of use cases. As products evolve and new features are introduced, I regularly update the module to ensure it stays current and continues to provide relevant, practical solutions.

Current version of this module as of writing this book is 3.7.

Module name: `vsadmin`

Installing the module (see Figure 5-6):

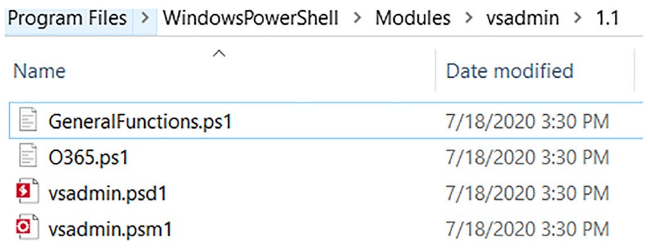
```
Install-Module -Name vsadmin
```



```
C:\temp> Install-Module -Name vsadmin

Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repository,
you sure you want to install the modules from 'PSGallery'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "N"):
```

***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.



```
Program Files > WindowsPowerShell > Modules > vsadmin > 1.1

Name                     Date modified
GeneralFunctions.ps1     7/18/2020 3:30 PM
O365.ps1                 7/18/2020 3:30 PM
vsadmin.psd1             7/18/2020 3:30 PM
vsadmin.psm1             7/18/2020 3:30 PM
```

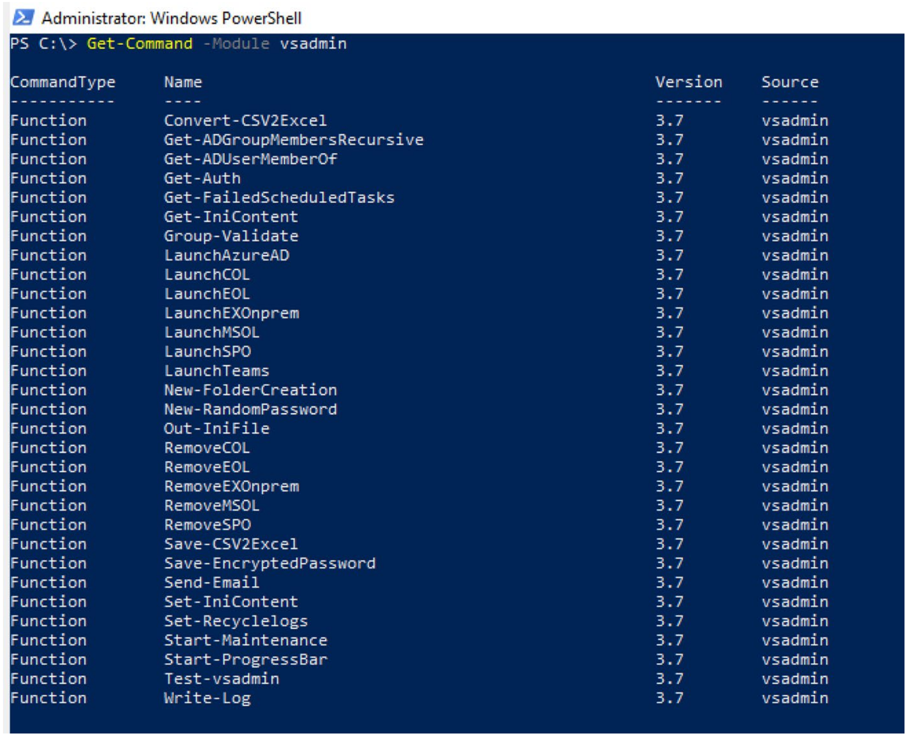
***Figure 5-7.** Showing files inside the vsadmin module*

As stated, you can import the module into 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
```



```
Administrator: Windows PowerShell
PS C:\> Get-Command -Module vsadmin

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Function        Convert-CSV2Excel                                  3.7        vsadmin
Function        Get-ADGroupMembersRecursive                        3.7        vsadmin
Function        Get-ADUserMemberOf                                 3.7        vsadmin
Function        Get-Auth                                           3.7        vsadmin
Function        Get-FailedScheduledTasks                           3.7        vsadmin
Function        Get-IniContent                                     3.7        vsadmin
Function        Group-Validate                                     3.7        vsadmin
Function        LaunchAzureAD                                      3.7        vsadmin
Function        LaunchCOL                                          3.7        vsadmin
Function        LaunchEOL                                          3.7        vsadmin
Function        LaunchEXOnprem                                     3.7        vsadmin
Function        LaunchMSOL                                         3.7        vsadmin
Function        LaunchSPO                                          3.7        vsadmin
Function        LaunchTeams                                        3.7        vsadmin
Function        New-FolderCreation                                 3.7        vsadmin
Function        New-RandomPassword                                 3.7        vsadmin
Function        Out-IniFile                                        3.7        vsadmin
Function        RemoveCOL                                          3.7        vsadmin
Function        RemoveEOL                                          3.7        vsadmin
Function        RemoveEXOnprem                                     3.7        vsadmin
Function        RemoveMSOL                                         3.7        vsadmin
Function        RemoveSPO                                          3.7        vsadmin
Function        Save-CSV2Excel                                     3.7        vsadmin
Function        Save-EncryptedPassword                             3.7        vsadmin
Function        Send-Email                                         3.7        vsadmin
Function        Set-IniContent                                     3.7        vsadmin
Function        Set-Recyclelogs                                    3.7        vsadmin
Function        Start-Maintenance                                  3.7        vsadmin
Function        Start-ProgressBar                                  3.7        vsadmin
Function        Test-vsadmin                                       3.7        vsadmin
Function        Write-Log                                          3.7        vsadmin
```

Figure 5-8. *Available vsadmin module commands*

Let's explore each of the function of the module to that you can utilize it in your scripts.

1. **Convert-CSV2Excel**

   The Convert-CSV2Excel function is designed to help you convert a CSV file to an Excel file without requiring Microsoft Excel to be installed on your machine. This is especially useful in environments where Excel isn't available or where you need to automate the conversion process as part of a script.

Usage:

Convert-CSV2Excel -CSVFile c:\csvfilepath\csvfile.
csv -ExcelFile c:\excelfilepath\excelfilepath.xlsx

Parameters:

- -CSVFile: Specifies the path to the CSV file that you want to convert
- -ExcelFile: Specifies the path where the resulting Excel file will be saved

2. **Get-ADGroupMembersRecursive**

The Get-ADGroupMembersRecursive function allows you to extract Active Directory group memberships recursively, meaning it can retrieve not only direct group members but also members of nested groups. This function can handle multiple groups simultaneously and retrieve specific user properties if needed, making it a powerful tool for managing and auditing AD group memberships.

Usage:

Get-ADGroupMembersRecursive -Groups "Test Nested Group1″ ,"Test Nested Group2"
Get-ADGroupMembersRecursive -Groups "Test Nested Group1″ ,"Test Nested Group2" -Properties Employeeid

Parameters:

- -Groups: Specifies one or more Active Directory groups to retrieve members from. Supports multiple group names.

- -Properties: Optional. Specifies which properties of the users or objects to retrieve. By default, it retrieves standard properties such as Name, SamAccountName, etc.

3. **Get-ADUserMemberOf**

   The Get-ADUserMemberOf function is a simple and effective way to check whether a specified user is a member of a particular Active Directory group. It returns True if the user is a member of the group (directly or indirectly) and False if they are not. This is useful for verifying user group memberships in automation scripts or access control processes.

   Usage:

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

   Parameters:

   - -User: Specifies the username or the Distinguished Name (DN) of the user you want to check
   - -Group: Specifies the name or Distinguished Name (DN) of the group you want to verify

4. **Get-Auth**

   The Get-Auth function simplifies the process of securely handling credentials within your PowerShell scripts. It retrieves a user's credentials from either an encrypted password file or an already encrypted password string, returning credentials in two formats: plain text and as a PSCredential object. This is particularly useful when working with APIs or PowerShell functions that require credential input, offering both security and flexibility.

Usage:

Option 1: Using encrypted password file

$cred = Get-Auth -UserId "sukhija@techwizard.
cloud" -PasswordFile "C:\data\password1.txt"
$pwd = $cred[0] ### Plain text password for use in
APIs or CSOM calls. $pscredential = $cred[1] ###
PSCredential object for PowerShell functions.

Option 2: Using encrypted password string

$cred = Get-Auth -UserId "sukhija@techwizard.
cloud" -Password "encryptedpassword" $pwd =
$cred[0] ### Plain text password for use in APIs
or CSOM calls. $pscredential = $cred[1] ###
PSCredential object for PowerShell functions.

- $cred[0] stores the plain text password, which can be used for CSOM (Client-Side Object Model) or API calls that require basic authentication.
- $cred[1] stores the credentials as a PSCredential object, useful for PowerShell functions that accept the PSCredential type.

5. **Get-FailedScheduledTasks**

The Get-FailedScheduledTasks function is designed to assist in monitoring and identifying failed tasks within the Windows Task Scheduler. It enables you to specify certain task paths (or folders) to focus on, providing a clear and concise way to track the health of scheduled tasks in your environment. This function is particularly useful for maintaining automated workflows and ensuring that critical scheduled tasks are running as expected.

Usage:

Get-FailedScheduledTasks -includepaths "Scheduled", "DevSolutions"

Parameters:

- -IncludePaths: Specifies the paths (or folders) within the Task Scheduler where you want to check for failed tasks. You can include multiple paths.

6. **Get-IniContent/Set-IniContent/Out-IniFile**

These three functions allow you to seamlessly read, modify, and write INI configuration files. By combining them, you can read INI data, update or add new values and then write the updated content back to the INI file in a structured, readable format.

Usage: Reading from an INI File

$readini = Get-IniContent $inifile

$vartest = $readini["initable"].value

- $inifile: Specifies the path to the INI file you want to read
- $readini: This variable holds the content of the INI file in a structured format, usually as a hashtable or similar object that you can easily access

Usage: Writing to an INI File

The Set-IniContent function allows you to modify or set values within specific sections of an INI file. It works by passing the section(s) and key-value pairs that need to be added or updated.

```
Set-IniContent -FilePath $configfile -Sections
$Sections -NameValuePairs @
{$PasswordKey=$getencpassword}
```

Parameters:

- **-FilePath**: Specifies the path to the INI file
- **-Sections**: Specifies the section (or sections) within the INI file to modify
- **-NameValuePairs**: A hashtable containing key-value pairs where the keys are the INI keys and the values are the new values to be written

Writing Changes Back to the INI File (Out-IniFile)

Once you have made modifications to the INI content, the Out-IniFile function is used to write the changes back to the INI file. This function allows you to control the format, encoding, and structure of the output file.

```
$ini = Set-IniContent -FilePath $configfile -Sections
$Sections -NameValuePairs @
{$PasswordKey=$getencpassword}
```

```
$ini | Out-IniFile $configfile -Pretty -Force -Encoding 'ASCII'
```

$ini: This holds the modified INI content after using Set-IniContent.

Out-IniFile: Writes the content back to the specified file.

- **-Pretty**: Formats the INI file in a readable way with proper indentation and spacing

- **-Force**: Ensures that the file is overwritten if it already exists
- **-Encoding**: Specifies the encoding of the file. In this example, it's set to ASCII

7. **Group-Validate**

   The Group-Validate function is designed to validate whether the provided user objects are valid groups within a given domain. It accepts a list of users, checks if they belong to a specific domain, and then tries to validate whether they are actual groups. Any invalid groups are logged to a report file.

   Usage:
   Group-Validate -User "domain\group1" -dom "domain" -greport "C:\logs\invalidGroups.txt"

   Parameters:

   - **$User**: A mandatory parameter, expects the user or group to be validated
   - **$dom**: A mandatory parameter, representing the domain to be used in group validation
   - **$greport**: Optional parameter, a file path where the invalid groups are logged

8. **Launch/Remove functions**

   The Launch functions are designed to connect you to various Office 365 services (e.g., Exchange Online, SharePoint Online, etc.) by importing the required PowerShell modules or establishing a session.

The Remove functions, on the other hand, help to clean up or disconnect the sessions after you're done working with these services.

Each pair of functions (e.g., LaunchEOL/RemoveEOL) is dedicated to a specific service and is prefixed with a name that differentiates it from similar on-premise commands to prevent conflicts in hybrid environments.

- `LaunchEOL/RemoveEOL` (Exchange Online)—prefixed as EOL

  LaunchEOL

  Get-EOLMailbox #Fetches mailboxes

  RemoveEOL
- `LaunchCOL/RemoveCOL` (Security and Compliance)—prefixed as COL
- `LaunchEXOnprem/RemoveEXOnprem` (for on-premise Exchange Server)
- `LaunchMSOL/RemoveMSOL` (MSonline Azure Active Directory)
- `LaunchSPO/RemoveSPO` (SharePoint online)

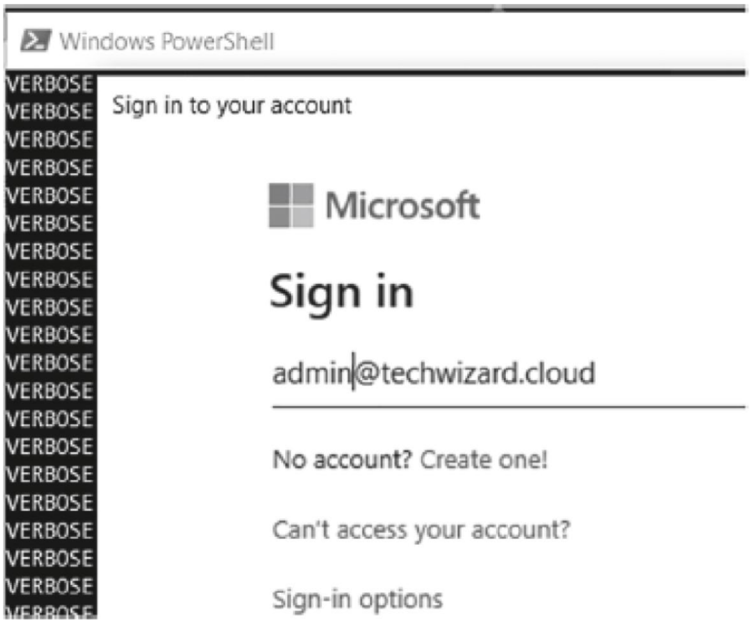
Why Prefix Functions for Office 365?

In hybrid environments, where both on-premise and cloud services are used, there can be command conflicts. For example, Get-Mailbox is used for both Exchange Online and on-premise Exchange, but the commands may differ in their behavior or requirements. By prefixing the commands (e.g., Get-EOLMailbox for Exchange Online), the functions avoid conflicts and make it clear which environment you're working with.

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:** `https://www.microsoft.com/en-ca/download/details.aspx?id=35588`
- **MSOnline Module (depreciated):** `www.powershellgallery.com/packages/MSOnline`

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.



Windows PowerShell

VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE
VERBOSE

Sign in to your account

Microsoft

Sign in

admin@techwizard.cloud

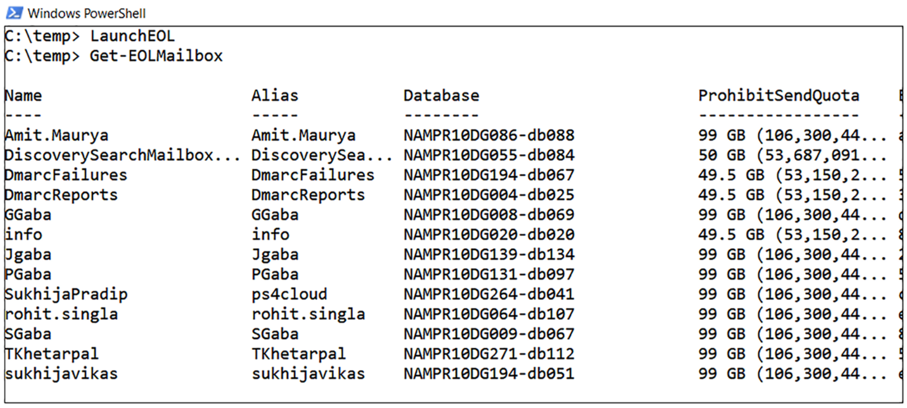
No account? Create one!

Can't access your account?

Sign-in options

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...
```

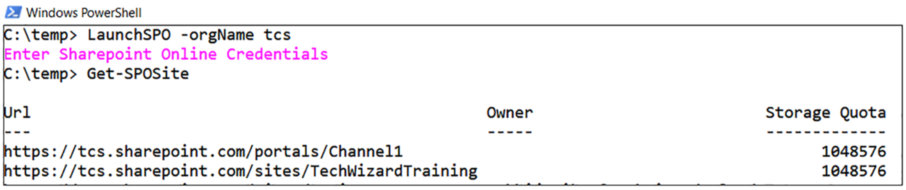
Figure 5-10. *Exchange Online Management Shell prefixed Get-MailBox command*

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.

In this latest vsadmin, **LaunchEOL** has also been updated to use **clientid/certificate** to do the APP authentication.

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
```



Windows PowerShell

```
C:\temp> LaunchSPO -orgName tcs
Enter Sharepoint Online Credentials
C:\temp> Get-SPOSite

Url                                                    Owner                          Storage Quota
---                                                    -----                          -------------
https://tcs.sharepoint.com/portals/Channel1                                                 1048576
https://tcs.sharepoint.com/sites/TechWizardTraining                                         1048576
```

***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
```

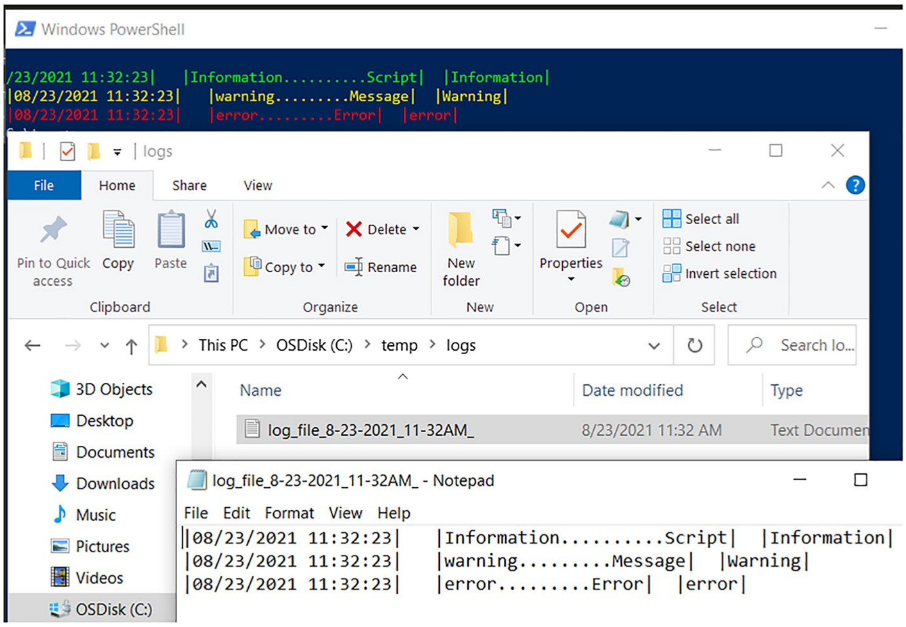
9. **Generic functions**

   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
```



```
/23/2021 11:32:23|    |Information.........Script|   |Information|
|08/23/2021 11:32:23|    |warning........Message|   |Warning|
|08/23/2021 11:32:23|    |error.........Error|   |error|
```

Windows PowerShell
logs
File Home Share View
Pin to Quick access, Copy, Paste, Move to, Delete, Copy to, Rename, New folder, Properties, Select all, Select none, Invert selection
Clipboard, Organize, New, Open, Select
This PC > OSDisk (C:) > temp > logs
Search lo...
3D Objects, Desktop, Documents, Downloads, Music, Pictures, Videos, OSDisk (C:)
Name, Date modified, Type
log_file_8-23-2021_11-32AM_ 8/23/2021 11:32 AM Text Documen
log_file_8-23-2021_11-32AM_ - Notepad
File Edit Format View Help

```
||08/23/2021 11:32:23|    |Information.........Script|   |Information|
|08/23/2021 11:32:23|    |warning........Message|   |Warning|
|08/23/2021 11:32:23|    |error.........Error|   |error|
```

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
```

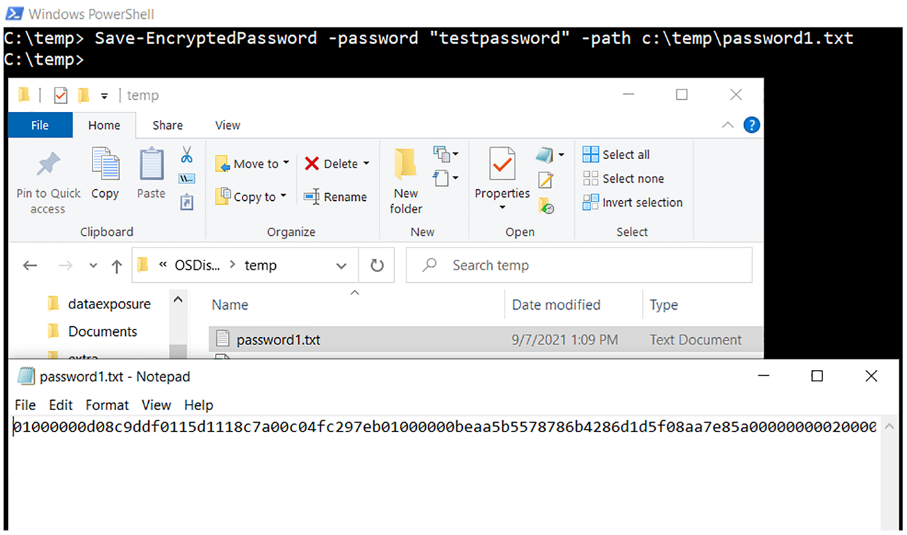
I will not get into the other functions that have been shared in previous chapters. I just wanted to show that they can all be used in this manner as well.

10. **Save-EncryptedPassword**

To securely store a password for later use in PowerShell scripts, such as for connecting to online services like Office 365 or Azure, you can use the Save-EncryptedPassword function. This function encrypts the password and saves it to a specified file, ensuring that sensitive credentials are not stored in plain text. Once the password is saved, it can be easily retrieved and used in your scripts to establish connections to various services. (see Figure 5-13 for the result):

Usage:

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



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

Name	Date modified	Type
password1.txt	9/7/2021 1:09 PM	Text Document

password1.txt - Notepad

```
01000000d08c9ddf0115d1118c7a00c04fc297eb01000000beaa5b5578786b4286d1d5f08aa7e85a00000000020000
```

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

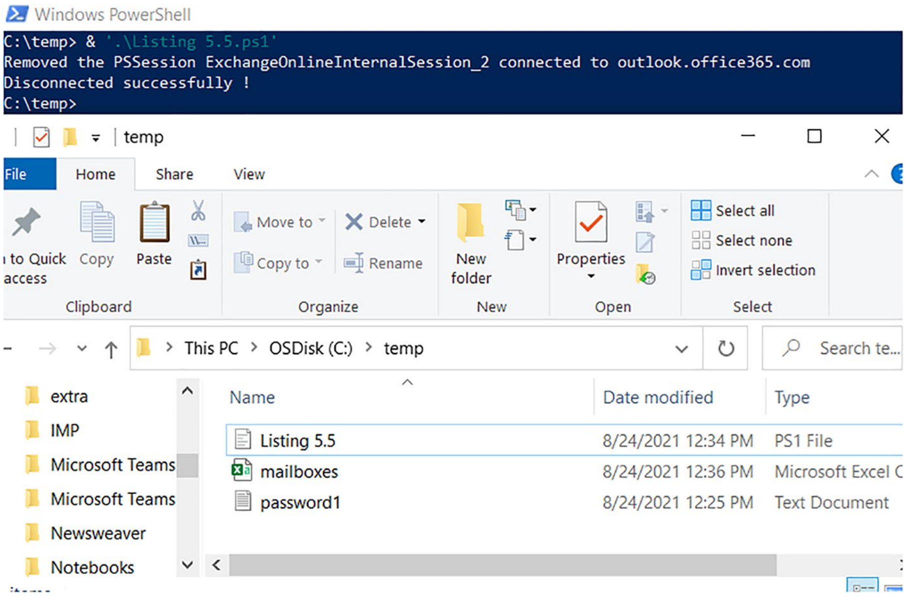
Parameters:

- **-Password:** The password you want to encrypt
- **-Path:** The file path where the encrypted password will be stored

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 from Saved File

```
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,Windo
wsEmailAddress,IssueWarningQuota, ProhibitSendQuota,ProhibitSen
dReceiveQuota #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
```



```
C:\temp> & '.\Listing 5.5.ps1'
Removed the PSSession ExchangeOnlineInternalSession_2 connected to outlook.office365.com
Disconnected successfully !
C:\temp>
```

Name	Date modified	Type
Listing 5.5	8/24/2021 12:34 PM	PS1 File
mailboxes	8/24/2021 12:36 PM	Microsoft Excel C
password1	8/24/2021 12:25 PM	Text Document

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

11. **`New-RandomPassword`**

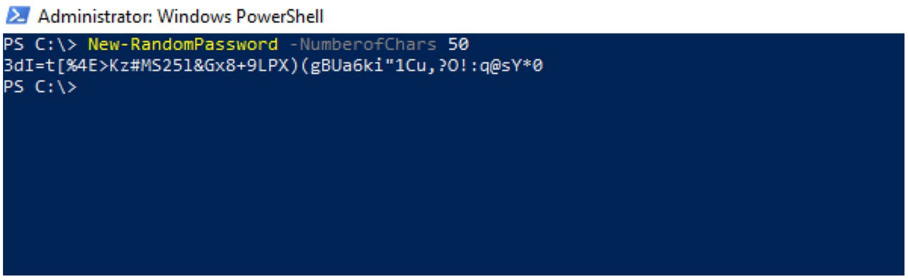
    The New-RandomPassword function is a handy tool for generating complex passwords directly from PowerShell, making it easier for system administrators to create secure passwords without relying on external tools or websites.

    See this example in Figure 5-15:

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

    You can add any number of chars to generate the password.

    ```
    New-RandomPassword - NumberofChars 50
    ```



```
PS C:\> New-RandomPassword -NumberofChars 50
3dI=t[%4E>Kz#MS251&Gx8+9LPX)(gBUa6ki"1Cu,?O!:q@sY*0
PS C:\>
```

Figure 5-15. *Showing the generation of a random password*

Summary

In this chapter, you explored the crucial role of modules in PowerShell scripting. Modules are essential components that extend the functionality of PowerShell, making it easier to manage and automate tasks across different products and systems. Without modules, scripting can be significantly limited.

You also introduced a specific cheat system administration module called **vsadmin**. This module includes a variety of functions and cmdlets that are commonly used in daily administrative tasks, helping streamline and automate routine activities.

Key Takeaways:

- **Modules as Essential Tools**: Modules are likened to batteries in PowerShell, providing the necessary functionality to script effectively across different environments.

- **vsadmin Module**: This module includes useful functions such as New-RandomPassword, Save-EncryptedPassword, and various Launch/Remove functions for managing Office 365 services. These tools are designed to simplify and enhance system administration tasks.

This chapter provides a foundation for understanding how to leverage PowerShell modules to boost productivity and efficiency in system administration and automation tasks.

CHAPTER 6

Alerting (Email)

Sending email notifications is a critical aspect of scripting, especially in automation scenarios where alerts need to be sent when errors occur or tasks complete. For instance, if you want to send an alert when your script encounters an error, or if you need to send bulk emails without relying on third-party email tools, PowerShell provides an easy-to-use cmdlet for this purpose: **Send-MailMessage**.

Introduced in PowerShell version 2, Send-MailMessage is a built-in cmdlet for sending emails. Below is an example of how it works:

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

-SmtpServer: This is the address of your SMTP server, through which the email will be sent.

-From: The sender’s email address.

-To: The recipient’s email address. Multiple recipients can be added by separating addresses with commas.

-Subject: The subject line of the email.

-Body: The content of the email message.

You can also expand this basic example to add more advanced features, such as including attachments, setting priorities, or using SSL for secure transmission. Here’s an updated version:

```
Send-MailMessage -SmtpServer “smtpserver” `
```

```
-From “DoNotReply@labtest.com” `
-To “sukhija@techwizard.cloud” `
-Subject “Error exception occurred” `
-Body “body of the message” `
-Priority High `
-Attachments “C:\Logs\errorlog.txt” `
-UseSsl
```

If you are still using **PowerShell 1.0** (which is highly unlikely given its age and the availability of newer versions), you can use the code in Listing 6-1. This method works on all versions of PowerShell, including v1, and utilizes the .NET System.Net.Mail namespace for sending emails. This approach was commonly used before PowerShell introduced native cmdlets like Send-MailMessage.

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

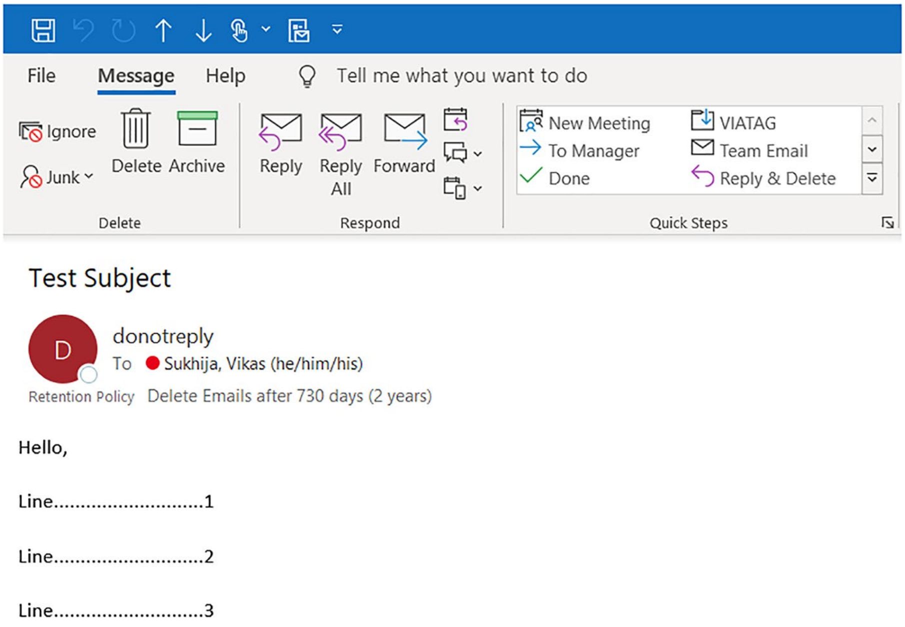
Formatting a Message Body

In many scenarios, sending a well-formatted email body is essential for better readability and professionalism, especially when conveying important information such as reports, error logs, or multiline messages. Instead of sending a simple one-liner email, you can format the body to include multiple lines and structured content.

Listing 6-2 demonstrates how to send a properly formatted email body and Figure 6-1 shows the resulted output. This approach allows you to create multiline messages using the here-string (@“”@) format, which makes your script cleaner and easier to modify.

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
```



Test Subject

donotreply

To Sukhija, Vikas (he/him/his)

Retention Policy Delete Emails after 730 days (2 years)

Hello,

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

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

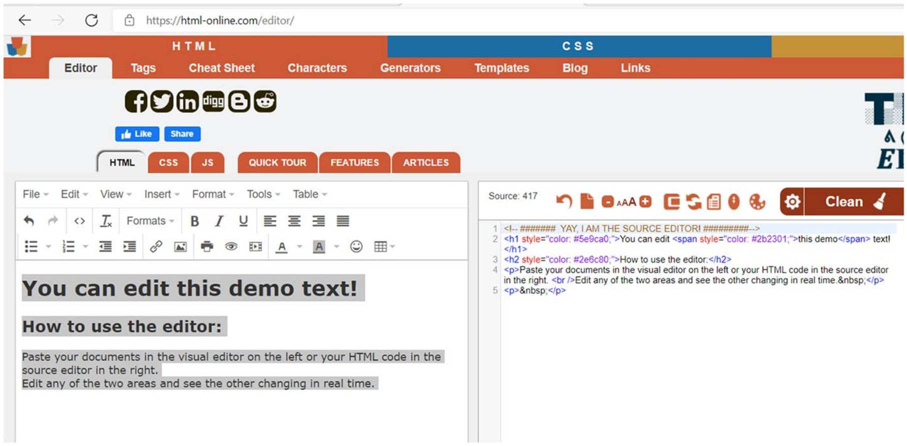
Line..........................3

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

Sending HTML

If you're looking to send more visually appealing emails with colors, fonts, and tables, but you're not familiar with HTML, there's a handy cheat tip you can use. Instead of manually writing the HTML, you can leverage an online HTML editor to easily design and generate the HTML code for you.

One such tool is the **HTML Online Editor**, which simplifies the process of creating professional HTML emails. You can access it at `https://html-online.com/editor/` (as shown in Figure 6-2). This editor allows you to visually design your email content and copy the generated HTML code directly into your PowerShell script.



You can edit this demo text!

How to use the editor:

Paste your documents in the visual editor on the left or your HTML code in the source editor in the right.
Edit any of the two areas and see the other changing in real time.

```
<!-- #######  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.&nbsp;</p>
<p>&nbsp;</p>
```

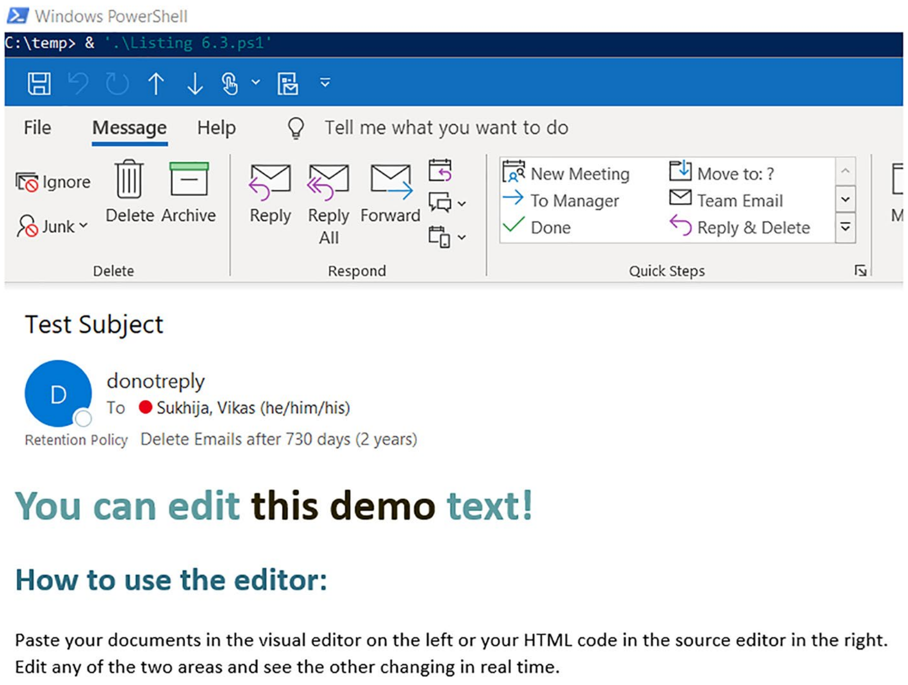
Figure 6-2. *The HTML Online Editor*

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.&nbsp;</p>
<p>&nbsp;</p>
"@
```

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



```
C:\temp> & '.\Listing 6.3.ps1'
```

Test Subject

donotreply
To Sukhija, Vikas (he/him/his)
Retention Policy Delete Emails after 730 days (2 years)

You can edit this demo text!

How to use the editor:

Paste your documents in the visual editor on the left or your HTML code in the source editor in the right. Edit any of the two areas and see the other changing in real time.

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

Sending Email—PowerShell Graph SDK

If you're working in a Microsoft 365 environment and want to send emails programmatically, the Microsoft Graph SDK provides a powerful and modern way to do so. The Microsoft Graph API allows you to interact with various Office 365 services, including Outlook, making it easy to send emails securely via PowerShell, even with rich features like attachments and HTML formatting.

By using the PowerShell Graph SDK, you gain several benefits over traditional email-sending methods:

- No need to configure SMTP servers.
- You can send emails using your Office 365 account.
- Leverage enhanced security features like OAuth 2.0 for authentication.

Install the graph SDK module from PowerShell Gallery.

Install-Module Microsoft.Graph

Register an Azure AD (Entra)Application: Set up an Azure AD application and configure the necessary API permissions (Mail.Send) for Microsoft Graph. This allows the script to authenticate and send emails on behalf of the account.

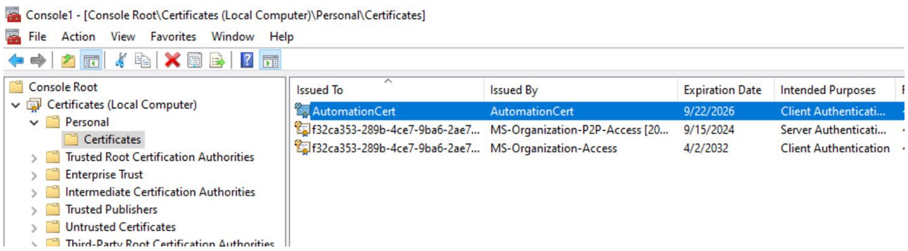
This should be application permission not delegated permission.

For authentication, type I recommend certificate-based authentication instead of client ID and client secret.

You can generate self-signed certificate using below powershell code.

New-SelfSignedCertificate -Subject 'CN=AutomationCert' -KeyLength 2048 -KeyUsageProperty All -KeyAlgorithm 'RSA' -HashAlgorithm 'SHA256' -Provider 'Microsoft Enhanced RSA and AES Cryptographic Provider' -NotAfter (Get-Date).AddYears(2)

Certificate will be generated under local machine context as shown in Figure 6-4.



Console1 - [Console Root\Certificates (Local Computer)\Personal\Certificates]

Console Root
- Certificates (Local Computer)
  - Personal
    - Certificates
  - Trusted Root Certification Authorities
  - Enterprise Trust
  - Intermediate Certification Authorities
  - Trusted Publishers
  - Untrusted Certificates
  - Third-Party Root Certification Authorities

Issued To	Issued By	Expiration Date	Intended Purposes
AutomationCert	AutomationCert	9/22/2026	Client Authenticati...
f32ca353-289b-4ce7-9ba6-2ae7...	MS-Organization-P2P-Access [20...	9/15/2024	Server Authenticati...
f32ca353-289b-4ce7-9ba6-2ae7...	MS-Organization-Access	4/2/2032	Client Authentication

Figure 6-4. *Generated certificate from PowerShell*

You can then export this certificate including private key and delete this certificate from the certificate store.

Save the certificate in some safe location.

You can import this certificate to the machine from where you want to use this graph SDK.

Import it in current user store so that only that user is able to access it and none of the admins of the machine is able to access.

You can further secure it on that machine by clicking export and select delete the private key so it is not exportable again from there (delete the exported certificate).

After that, just export the certificate (.cer) and upload it to the registered Azure APP.

Now connect the Graph SDK module on the machine where you installed it and send email as shown in Listing 6-4.

Listing 6-4. Sending Email from PowerShell SDK

```
Connect-MgGraph -ClientId $MgGClientID -CertificateThumbprint
$ThumbPrint -TenantId $TenantName -NoWelcome #
connection command
$message = @{
      message = @{
            subject = "Automation Script”
            body = @{
                  contentType = "Text"
                  content = "Testing"
            }
            toRecipients = @(
                  @{
                        emailAddress = @{
                              address = "SV1@techwizard.cloud"
                        }
```

```
                }
            )
            ccRecipients = @(
                @{
                    emailAddress = @{
                        address = "SV2@TechWizard.cloud"
                    }
                }
            )
        }
        saveToSentItems = "false"
}

Send-MgUserMail -UserId $From -BodyParameter $message
```

Summary

In this chapter, you learned various ways to send emails using PowerShell. Whether you're working with simple, one-liner emails or crafting complex HTML-formatted messages, these techniques are essential for automating communication in real-world scenarios. You also explored both traditional methods, like Send-MailMessage, and modern approaches, such as leveraging the **Microsoft Graph SDK** for Office 365 environments.

Key Takeaways

- **Basic Email Sending**: You learned how to send a simple email using the Send-MailMessage cmdlet and the legacy approach for PowerShell v1.
- **Formatted Emails**: You saw how to send properly formatted, multiline email messages using here-strings and how to generate HTML emails for more sophisticated, visually appealing communications.

- **PowerShell Graph SDK**: By leveraging the power of Microsoft Graph, you can send emails securely and flexibly in Office 365 environments without worrying about SMTP server configuration.

You can use this knowledge in real-world scenarios to

- **Send Bulk Emails**: Automate communication by sending bulk emails for newsletters, notifications, or updates without needing third-party email marketing tools.
- **Send Email Alerts**: Incorporate email notifications into your scripts and automation workflows to alert you when a task or process fails. This is particularly useful for monitoring systems and error reporting, ensuring you stay informed without manual oversight.

With the skills covered in this chapter, you're now equipped to automate email communication efficiently using PowerShell, whether it's for day-to-day operations or critical system alerts.

CHAPTER 7

Error Reporting

For successful scripting, **error reporting** is a must-have. When errors occur, it's essential to notify the administrator or owner of the automation process so they can address the issue quickly. PowerShell provides several ways to handle error reporting, and one of the most common is by using the **$Error** variable. This variable stores all the errors that have occurred in the current session, allowing you to access and log them effectively.

In addition to logging errors, it's often necessary to send alerts or notifications, such as via email, to ensure someone is informed when an issue arises. Below are some **cheat code examples** to show how you can implement error reporting using PowerShell. These examples demonstrate both logging errors and sending them via email.

Reporting Errors Through Email

Error reporting is crucial for ensuring that script failures are caught and dealt with in a timely manner. Below, I have provided a method to send errors via email when they occur in a PowerShell script.

In PowerShell, **$Error** is a default variable that stores all errors encountered during the session.

In Listing 7-1, we are checking if $Error is not null, and if so, send the latest error by email. After sending the email, $Error.Clear() is used to clear the error array, ensuring that if the script runs in a loop or multiple iterations, old errors won't be repeatedly sent if no new ones occur.

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_7

Listing 7-1. Sending Errors via Email

```
$from = "donotreply@lab.com"
$to="vikas@lab.com"
$subject = "Error has occured"
$smtpServer="smtp.lab.com"
if ($error)
      {
Send-MailMessage -SmtpServer $smtpserver -From $from -To
$to  -Subject $subject -Body $error[0].ToString()
$error.clear()
       }
```

Explanation:

- $error[0] sends the most recent error from the $Error array.
- $error.Clear() ensures that old errors are cleared, preventing them from being sent again in case the script loops or reruns.

However, Listing 7-1 only sends the last error, and $Error is actually an array containing all errors in the session. If you need to send the **full error log**, you can't directly use Send-MailMessage because it doesn't handle arrays or multiple error messages effectively.

To overcome this, you can utilize the Send-Email function from the **vsadmin** module (covered in the modules chapter of this book). The Send-Email function has been designed to handle arrays and format them properly for email content. This allows you to send the complete error list or multiple errors in a readable format.

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've imported the Send-Email function from Listing 7-2, you can use it to send the entire $Error array via email. This method ensures that all errors encountered during the script execution are properly captured and reported, improving visibility of potential issues.

The code in Listing 7-3 is nearly identical to Listing 7-1, but instead of using the built-in Send-MailMessage, it leverages the more flexible Send-Email function from the **vsadmin** module to handle sending the full error array.

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()
       }
```

By utilizing the Send-Email function, you can ensure that **all errors** are sent in one email, providing more comprehensive error reporting and reducing the risk of missing important issues during script execution.

Logging Everything Including Errors

To enhance your script with better logging, PowerShell offers **transcript logging** through the built-in cmdlets Start-Transcript and Stop-Transcript. This approach captures the output, commands, and errors that occur during script execution and logs them into a file. By default, the transcript is stored in the user's **My Documents** folder, but you can customize the location using the -Path parameter.

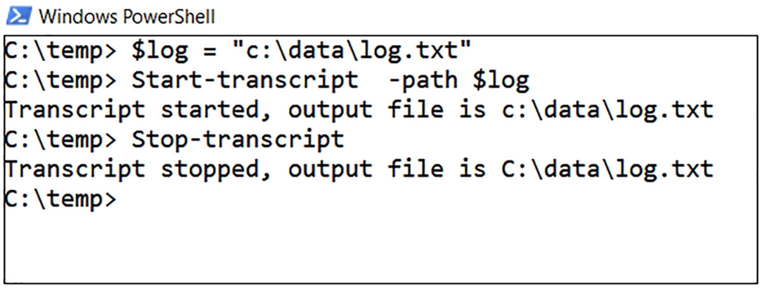
Transcript Logging Overview:

- **Start-Transcript**: Begins recording everything that happens in the script, including commands and output
- **Stop-Transcript**: Stops the recording and saves the transcript to a specified location or the default folder

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

Example: Using Transcript Logging

Here's how you can include transcript logging in your script, with a custom path to store the log as shown in Figure 7-1.



Windows PowerShell

```
C:\temp> $log = "c:\data\log.txt"
C:\temp> Start-transcript  -path $log
Transcript started, output file is c:\data\log.txt
C:\temp> Stop-transcript
Transcript stopped, output file is C:\data\log.txt
C:\temp>
```

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

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

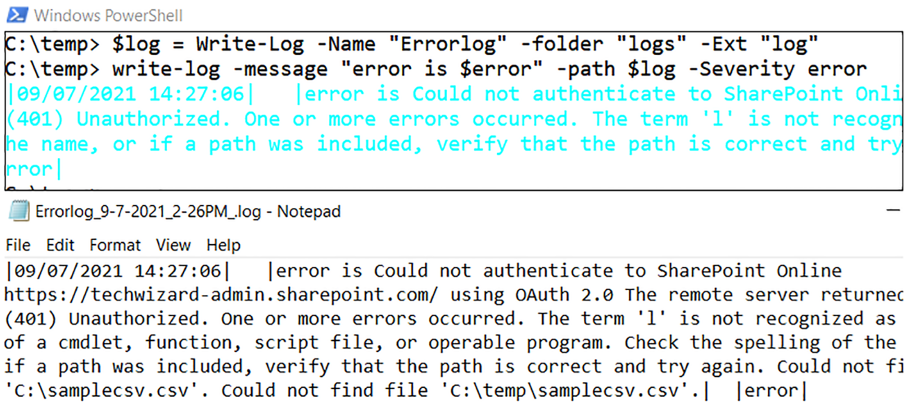
Logging Errors to a Text File

Logging errors to a text file is a simple yet effective way to keep a record of issues encountered during script execution. I'll show you how to implement a Write-Log function that can be reused across scripts to log errors or other important information in a structured manner.

The function allows you to customize the log file's name, folder, and extension, while also controlling the severity of the messages logged. This can be particularly useful for separating **informational**, **warning**, and **error** logs.

Below Write-Log function from vsadmin module can be utilized as shown below in Figure 7-2 to log errors.

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



```
C:\temp> $log = Write-Log -Name "Errorlog" -folder "logs" -Ext "log"
C:\temp> write-log -message "error is $error" -path $log -Severity error
|09/07/2021 14:27:06|    |error is Could not authenticate to SharePoint Onli
(401) Unauthorized. One or more errors occurred. The term 'l' is not recogn
he name, or if a path was included, verify that the path is correct and try
rror|
```

Errorlog_9-7-2021_2-26PM_.log - Notepad
File Edit Format View Help

```
|09/07/2021 14:27:06|    |error is Could not authenticate to SharePoint Online
https://techwizard-admin.sharepoint.com/ using OAuth 2.0 The remote server returnec
(401) Unauthorized. One or more errors occurred. The term 'l' is not recognized as
of a cmdlet, function, script file, or operable program. Check the spelling of the
if a path was included, verify that the path is correct and try again. Could not fi
'C:\samplecsv.csv'. Could not find file 'C:\temp\samplecsv.csv'.|   |error|
```

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

Try Catch

Using Try and Catch blocks in PowerShell is a more structured way to handle exceptions compared to relying solely on the $error variable. It allows for precise control over error handling, making scripts more robust and flexible. Here's how you can enhance error handling with Try/Catch, logging, and notifications.

The Try block contains the code that might throw an error, and if an error occurs, the Catch block is triggered to handle that error. This approach is especially useful when you want to log or handle specific types of errors.

Example is shown in Listing 7-4.

Listing 7-4. Try Catch Exception Handling

```
$smtpserver = "smtpserver"
$erroremail = "reports@labtest.com"
$from = "DoNotRespond@labtest.com"
try {
    # Code that might throw an error
    Get-Item "C:\InvalidPath"
} catch {
    $exception = $_.Exception.Message
    # Handling the error if an exception occurs
    Write-Error "An error occurred: $_"
    Send-MailMessage -SmtpServer $smtpserver -From $from -To
    $erroremail -Subject "Error -Message" -Body $exception
}
```

You can add Send-MailMessage or send-mail cmdlet under catch to send exception on email.

Summary

In this chapter, you learned various methods for error reporting in PowerShell, helping ensure that issues are properly captured, logged, and communicated:

1. **Reporting Errors**: We explored how to handle errors using the $error variable and send them via email using the built-in Send-MailMessage cmdlet or a custom Send-Email function.

2. **Logging Errors to a File**: You learned how to utilize the Write-Log function to create detailed logs, allowing you to store error messages in text files, complete with timestamps and severity levels.
3. **Email Notifications**: We covered how to automatically notify administrators or process owners by sending error details via email when something goes wrong.
4. **Session Capture with Start-Transcript**: Finally, we demonstrated how to capture the entire PowerShell session using the Start-Transcript and Stop-Transcript cmdlets, providing a complete record of script execution for later review.

These techniques, when combined with Try/Catch for exception handling, make your scripts more reliable, ensuring that any errors encountered are appropriately logged, reported, and handled.

CHAPTER 8

Reporting

Reports are essential tools in the day-to-day life of a system administrator. They not only help in presenting information to stakeholders, such as managers, but also serve as valuable resources for evaluating and improving work processes. Reports come in various formats like CSV, HTML, Excel, and more.

The **CSV (comma-separated values)** format is the most commonly used because of its versatility. CSV files can easily be imported into applications such as Excel, making it a universal reporting option.

CSV Report

PowerShell provides the Export-CSV cmdlet, a powerful and flexible tool for exporting data to a CSV file. Using this cmdlet, administrators can create custom reports by piping output from other commands like Get-Mailbox or Get-ADUser. The Export-CSV cmdlet makes it easy to take information from the system and save it in a structured way.

Below is an example showing how to export attributes of users' mailboxes from Exchange Server. This script selects key properties such as Name, Identity, WindowsEmailAddress, Database, ProhibitSendQuota, ProhibitSendReceiveQuota, and IssueWarningQuota and exports them into a CSV file.

V. Sukhija, *PowerShell Fast Track*, https://doi.org/10.1007/979-8-8688-1157-9_8

```
Get-Mailbox -ResultSize unlimited | Select Name,identity, Windows
EmailAddress,Database,ProhibitSendQuota,ProhibitSendReceiveQuota,
IssueWarningQuota | export-csv c:\mailboxes.csv -notypeinfo
```

This example can be run in the Exchange Management Shell, and it exports the selected mailbox properties to a CSV file located at C:\ mailboxes.csv.

Complex CSV Reports

Sometimes, the requirements for a report can be more complex. You might need to combine data from multiple sources, such as combining attributes from both Active Directory and Exchange Server. In such cases, a simple Get, Select, and Export-CSV approach is not enough.

For instance, consider a scenario where you have a list of users in a text file, and you want to export their Active Directory and Exchange attributes into a single CSV report. Here, additional scripting is required to gather and format the data appropriately before exporting.

Scenario: Exporting User Attributes from Active Directory and Exchange

In this example, we want to export the following attributes:

- **Exchange Server:** Name, Identity, WindowsEmailAddress, Database, ProhibitSendQuota, ProhibitSendReceiveQuota, and IssueWarningQuota
- **Active Directory:** EmployeeID, City (l), and Country (c)

We'll begin by reading the list of users from a text file, fetching their properties from both Active Directory and Exchange, and then combining that information into a CSV file. This is shown in Listing 8-1 and result in Figure 8-1.

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`:

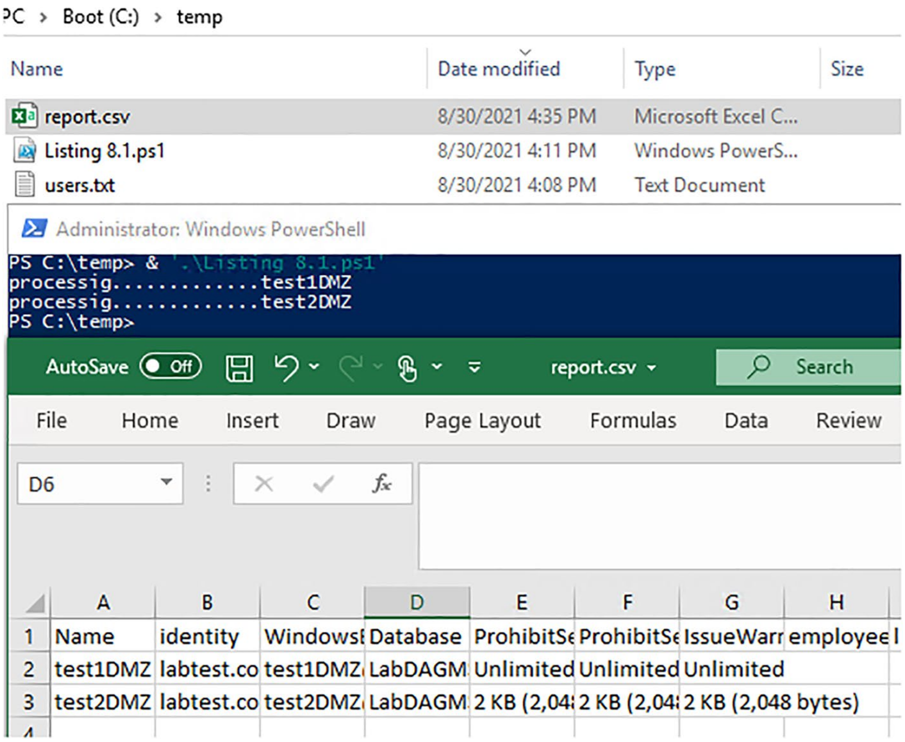
```
LaunchEXOnprem -ComputerName ExchangeServer
```

For Active Directory, you can use the following:

```
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.ProhibitSend
  ReceiveQuota
  $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
```



PC › Boot (C:) › temp

Name	Date modified	Type	Size
report.csv	8/30/2021 4:35 PM	Microsoft Excel C...	
Listing 8.1.ps1	8/30/2021 4:11 PM	Windows PowerS...	
users.txt	8/30/2021 4:08 PM	Text Document	

Administrator: Windows PowerShell

```
PS C:\temp> & '.\Listing 8.1.ps1'
processig............test1DMZ
processig............test2DMZ
PS C:\temp>
```

	A	B	C	D	E	F	G	H
1	Name	identity	Windows	Database	ProhibitSe	ProhibitSe	IssueWarr	employee I
2	test1DMZ	labtest.co	test1DMZ	LabDAGM	Unlimited	Unlimited	Unlimited	
3	test2DMZ	labtest.co	test2DMZ	LabDAGM	2 KB (2,04	2 KB (2,04	2 KB (2,048 bytes)	
4								

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

Another important aspect of CSV reporting is the ability to export **multivalued attributes**. Multivalued attributes contain multiple entries or values in a single field. This is particularly common when working with attributes like distribution groups, email recipients, or security groups in systems like **Exchange Server** or **Active Directory**.

When exporting such attributes, it is important to handle them correctly so that the data remains readable and usable. In PowerShell, we can use custom expressions within Select-Object to format multivalued attributes as needed.

Example: Extracting Multivalued Recipients from Exchange Tracking Logs

In Exchange, the **recipients** field is a multivalued attribute in the message tracking logs. When querying the logs, this field can contain one or more email addresses per message and exporting it directly may result in values that are difficult to parse in a CSV file.

To handle this, we can create a custom expression that joins the multivalued attributes into a readable format. The following expression ensures that recipients are extracted and formatted as a single string, with each recipient separated by a comma (you can use any other separator as well instead of comma).

```
@{Name="Recipients";Expression={$_.recipients -join ","}}
```

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

Listing 8-2. Example Code Showing How to Export Multivalued 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 -join ","}} | `
export-csv c:\track.csv
```

Excel Reporting

While CSV reports are sufficient for most purposes, there are situations where more polished and user-friendly formats are required—especially when sharing data with management or nontechnical stakeholders. **Excel** provides a highly versatile and visually appealing format for presenting data, offering features like formatting, charts, and pivot tables that make it easy to analyze and present information.

Converting data from PowerShell to Excel directly can be a more efficient and professional way of generating reports. Below, we will explore two methods for creating Excel reports from PowerShell scripts.

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

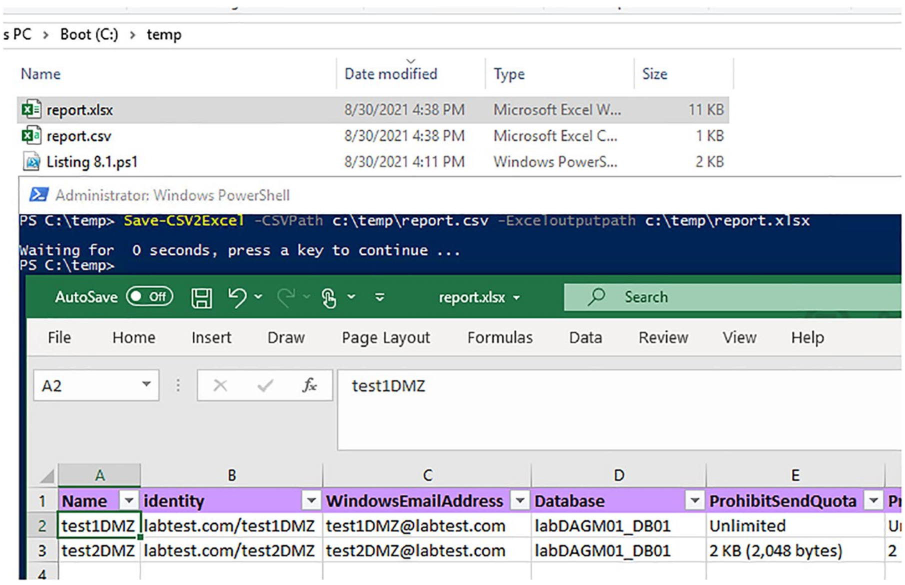
```
Function Save-CSV2Excel
{
  [CmdletBinding()]
  Param(
    [Parameter(Mandatory = $true,Position = 1)]
    [ValidateScript({

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
```



s PC › Boot (C:) › temp

Name	Date modified	Type	Size
report.xlsx	8/30/2021 4:38 PM	Microsoft Excel W...	11 KB
report.csv	8/30/2021 4:38 PM	Microsoft Excel C...	1 KB
Listing 8.1.ps1	8/30/2021 4:11 PM	Windows PowerS...	2 KB

Administrator: Windows PowerShell

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

Waiting for  0 seconds, press a key to continue ...
PS C:\temp>
```

A2 test1DMZ

	A	B	C	D	E	
1	Name	identity	WindowsEmailAddress	Database	ProhibitSendQuota	Pr
2	test1DMZ	labtest.com/test1DMZ	test1DMZ@labtest.com	labDAGM01_DB01	Unlimited	Ur
3	test2DMZ	labtest.com/test2DMZ	test2DMZ@labtest.com	labDAGM01_DB01	2 KB (2,048 bytes)	2
4						

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

One of the most popular and powerful PowerShell modules available in the PowerShell Gallery is the **ImportExcel** module. This module allows you to create and manipulate Excel files without needing to have Excel installed on the machine. This is a significant advantage when running scripts on servers or environments where installing Excel is not feasible or recommended.

The **ImportExcel** module simplifies the process of converting data directly into Excel files, allowing you to export variables, tables, or other forms of structured data into neatly formatted Excel spreadsheets.

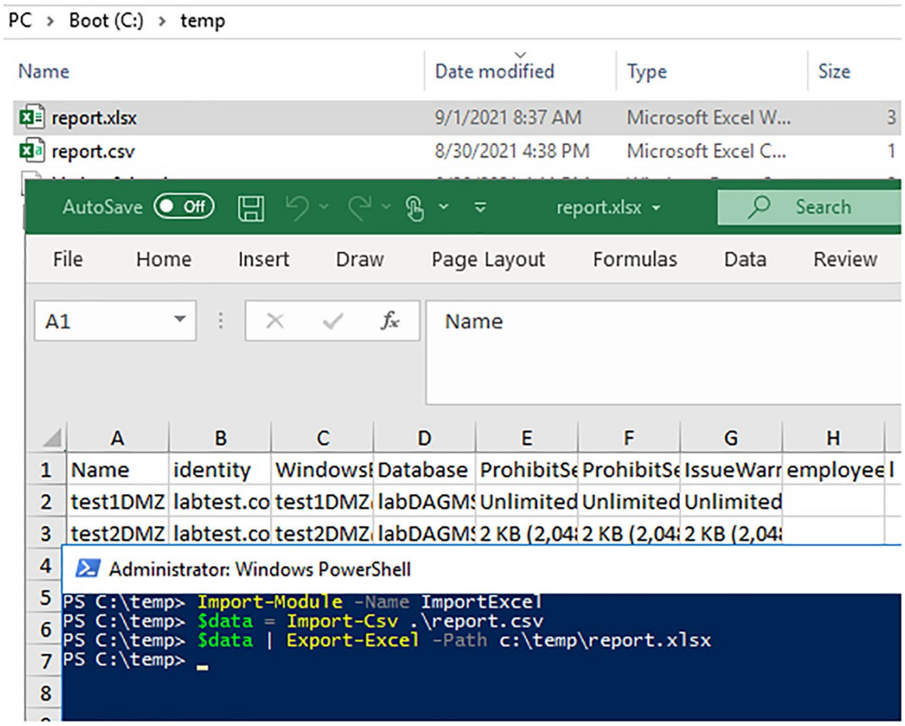
Installing the ImportExcel Module

To use the **ImportExcel** module, you first need to install it from the PowerShell Gallery. You can do this by running the following command:

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
```



PC > Boot (C:) > temp

Name	Date modified	Type	Size
report.xlsx	9/1/2021 8:37 AM	Microsoft Excel W...	3
report.csv	8/30/2021 4:38 PM	Microsoft Excel C...	1

	A	B	C	D	E	F	G	H
1	Name	identity	WindowsI	Database	ProhibitSe	ProhibitSe	IssueWarr	employee
2	test1DMZ	labtest.co	test1DMZ	labDAGMS	Unlimited	Unlimited	Unlimited	
3	test2DMZ	labtest.co	test2DMZ	labDAGMS	2 KB (2,04	2 KB (2,04	2 KB (2,04	

Administrator: Windows PowerShell

```
PS C:\temp> Import-Module -Name ImportExcel
PS C:\temp> $data = Import-Csv .\report.csv
PS C:\temp> $data | Export-Excel -Path c:\temp\report.xlsx
PS C:\temp> _
```

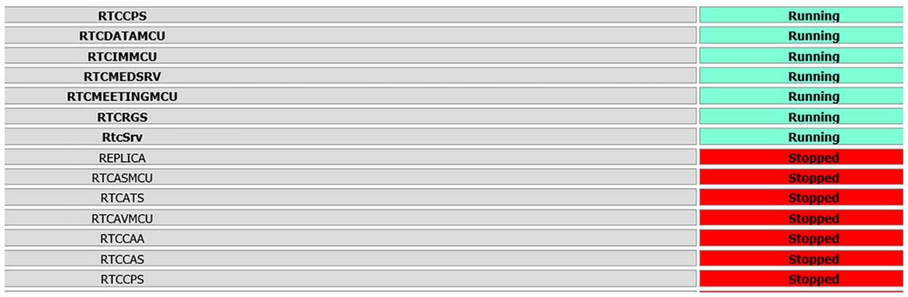
Figure 8-3. *Using the ImportExcel module*

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

HTML Reporting

HTML dashboards can be an effective way to visually represent the status of system services or infrastructure components. By using PowerShell, it's possible to generate real-time dashboards that display traffic light-style indicators: red when a service is down and green when it's running smoothly. This approach simplifies monitoring, enabling IT administrators to assess system health at a glance.

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>BackupInProgress</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 explored the essential topic of **reporting** in PowerShell. You learned how to generate reports in three common formats: **CSV**, **HTML**, and **Excel**, which are widely used across various systems for presenting data in a clear, accessible manner. CSV is a universal format, suitable for most reporting needs, while HTML allows you to create interactive and visually rich dashboards. The **ImportExcel** module further extends reporting capabilities, enabling you to generate Excel reports without needing Excel installed on the system.

By mastering these reporting techniques, you can provide valuable, actionable data to managers and stakeholders, ensuring that the information is presented in a format that's both professional and easy to understand.

CHAPTER 9

Miscellaneous Keywords

In this chapter, we will dive into some key PowerShell keywords that enable efficient data manipulation—a fundamental aspect of scripting and automation. Mastering these commands will allow you to perform tasks such as string manipulation, data comparison, and pattern matching, which are vital for processing information in scripts.

We will explore the following keywords:

- **Split**: Breaks a string into an array of substrings based on a specified delimiter, making it easier to handle and analyze data chunks
- **Replace**: Substitutes occurrences of one value within a string with another, an essential function for cleaning or transforming data
- **Select-String**: Searches for text patterns within files or strings, allowing you to locate specific information, like how you might use grep in other languages
- **Compare-Object**: Compares two sets of objects, highlighting differences or similarities, which is crucial for tracking changes or comparing datasets

These keywords are powerful tools for manipulating data and searching within files or collections. They are commonly used in real-world automation tasks, such as text parsing, log analysis, or comparing configuration states.

Split

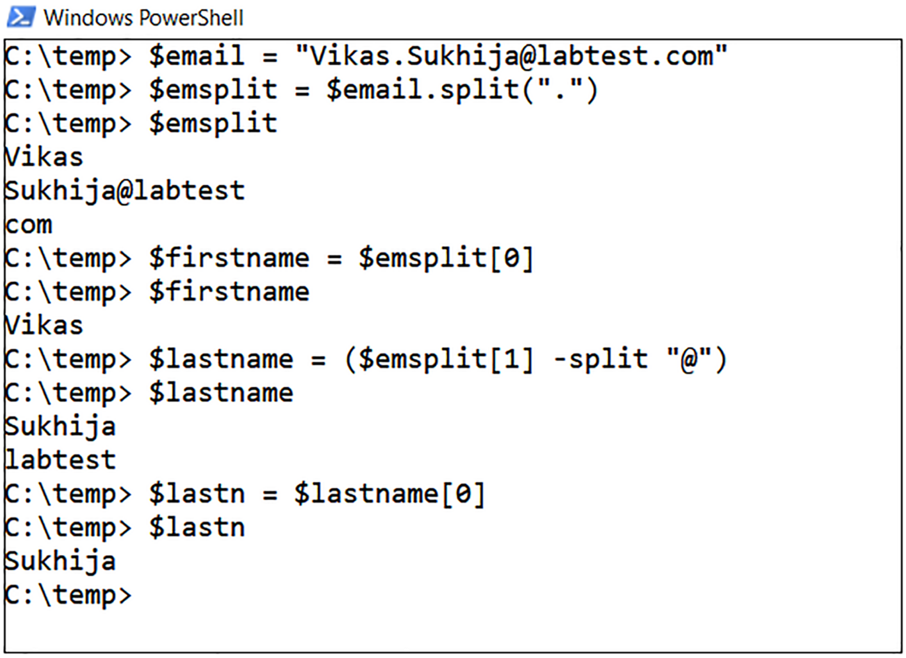
The **Split** keyword is incredibly useful for extracting data from strings. Whether you're pulling out an email address or parsing other structured text, Split allows you to break the string at a specified character and convert it into an array. Let's look at an example to see this in action.

In this example, we'll extract the first name and last name from an email address. You'll split the email string at the dot (.) to isolate the first name, then further split at the @ symbol to extract the last name.

```
$email = "Vikas.Sukhija@labtest.com"
$emsplit = $email.split(".")
$firstname = $emsplit[0]
$lastname = ($emsplit[1] -split "@")
$lastn = $lastname[0]
$emsplit[0] and $lastname[0]
```

In this example, the email string is first split at the dot (.), resulting in an array where the first element ($emsplit[0]) holds the first name. To get the last name, we split the second part of the email ($emsplit[1]) at the @ character, and the first element of that split becomes the last name.

Figure 9-1 illustrates this step-by-step process, helping you visualize how the Split operation works.



```
C:\temp> $email = "Vikas.Sukhija@labtest.com"
C:\temp> $emsplit = $email.split(".")
C:\temp> $emsplit
Vikas
Sukhija@labtest
com
C:\temp> $firstname = $emsplit[0]
C:\temp> $firstname
Vikas
C:\temp> $lastname = ($emsplit[1] -split "@")
C:\temp> $lastname
Sukhija
labtest
C:\temp> $lastn = $lastname[0]
C:\temp> $lastn
Sukhija
C:\temp>
```

Figure 9-1. *Showing the Split operation*

Replace

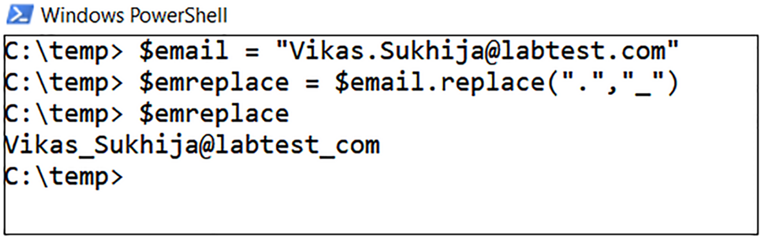
Another useful keyword is **Replace**. Instead of splitting a string, Replace allows you to substitute specific parts of a string with new content. This can be particularly useful when you need to update or modify the structure of text data.

For example, imagine you need to replace the dot (.) in an email address with an underscore (_) to create a modified version, such as for a secondary address.

Here's the PowerShell code to achieve this:

You can use this code and see the result in Figure 9-2:

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



```
C:\temp> $email = "Vikas.Sukhija@labtest.com"
C:\temp> $emreplace = $email.replace(".","_")
C:\temp> $emreplace
Vikas_Sukhija@labtest_com
C:\temp>
```

Figure 9-2. *Showing a Replace operation*

Select-String

The **Select-String** keyword is an incredibly powerful tool for searching within files. You can use it to locate specific strings in one or more files, making it ideal for tasks such as searching through logs or large datasets.

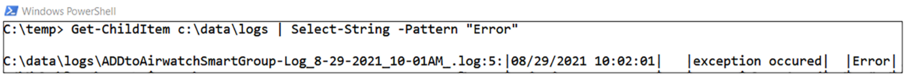
One practical use I have relied on many times is finding the exact date and time of an operation from a large set of log files. While others may spend hours manually scanning files, Select-String makes this process quick and efficient.

For instance, say you have a folder full of log files, and you need to find only the files where the word "Error" appears. The following one-liner will do just that:

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

This command retrieves all the files from the logs folder and searches each file for the word "Error." When found, it displays the file name along with the matching string.

You can see how this Select-String operation extracts the file name containing the error in Figure 9-3.



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

C:\data\logs\ADDtoAirwatchSmartGroup-Log_8-29-2021_10-01AM_.log:5:|08/29/2021 10:02:01|   |exception occured|  |Error|
```

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

Compare-Object

The **Compare-Object** keyword (or its alias Compare) is highly efficient for comparing two sets of data, whether they are files or arrays. It provides a faster and more reliable alternative to looping through arrays manually. One common use is comparing members of Active Directory groups with entries in a file or another group, identifying differences, and synchronizing the two sets.

For instance, you can fetch members from one group and compare them with a list of user IDs from a text file. This way, you can add only the missing members instead of processing all members. Below is an example demonstrating how to add members from one group to another (Listing 9-1).

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 explored several powerful keywords in PowerShell that are essential for data manipulation and transformation tasks. These keywords—**Split, Replace, Select-String,** and **Compare-Object**—enable you to efficiently manage and modify data. Whether you need to extract specific details from strings, search for patterns in files, or compare and synchronize large datasets, these tools streamline the process. By mastering these commands, you can automate operations even when the input data comes in unexpected formats, making your scripts more dynamic and adaptable to real-world scenarios.

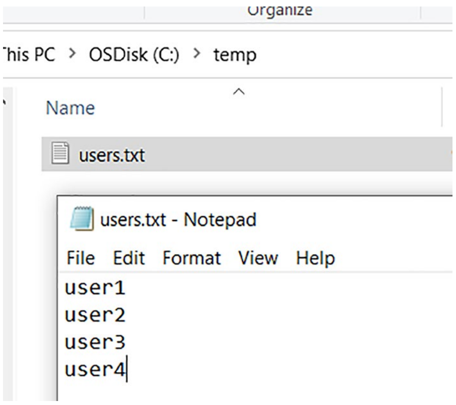
CHAPTER 10

Gluing It All Together

Welcome to the final chapter of this book! In this chapter, I will guide you through creating a practical script that applies the knowledge and techniques you've learned so far. Additionally, I will share some valuable cheat codes from various products that can simplify your daily automation tasks.

Scenario Overview

Imagine this scenario: You receive a text file from your HR system (see Figure 10-1), which contains a list of account names. Your task is to add these accounts to an Active Directory (AD) group. By doing so, you will grant them access to a file share that has permissions for that AD group or push an application to their devices based on their membership in the group.



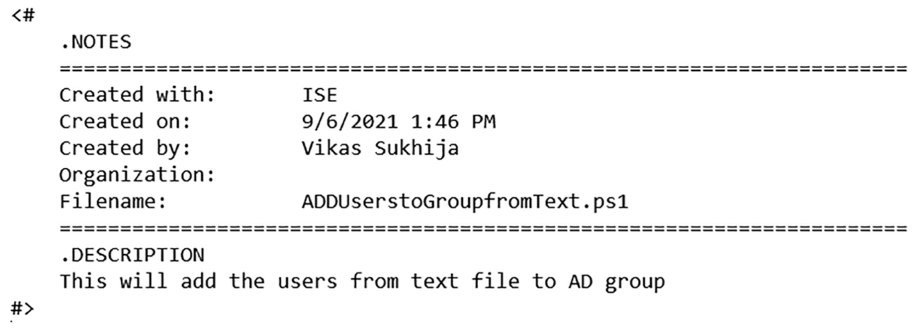
```
user1
user2
user3
user4
```

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

This is a common administrative task, but with automation, you can streamline the process and eliminate manual effort. We'll walk through creating a script that automates adding users to the AD group, ensuring that you can handle this task efficiently and with minimal risk of errors.

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
    }
  }
}
```

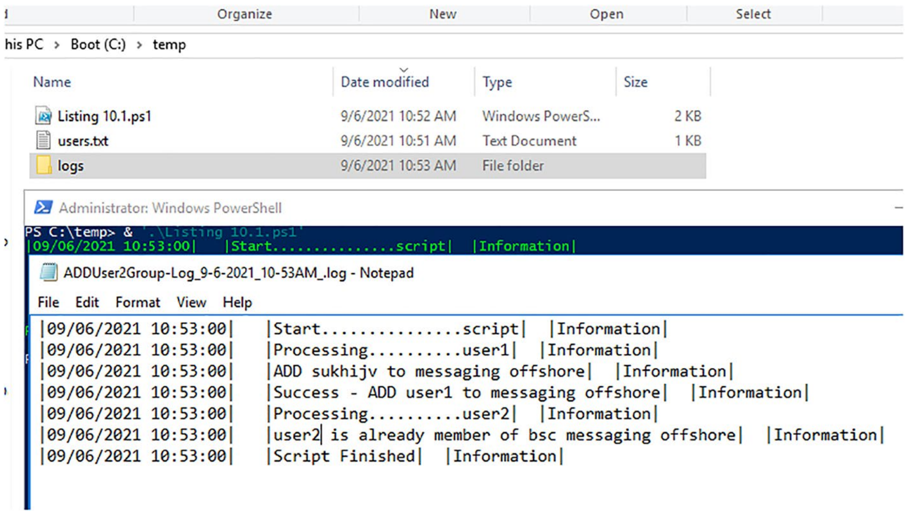
Step 5: Recycle logs or clean up the sessions (see Listing 10-1)

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 it 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.


Organize
New
Open
Select
his PC > Boot (C:) > temp

Name	Date modified	Type	Size
Listing 10.1.ps1	9/6/2021 10:52 AM	Windows PowerS...	2 KB
users.txt	9/6/2021 10:51 AM	Text Document	1 KB
logs	9/6/2021 10:53 AM	File folder	

Administrator: Windows PowerShell

```
PS C:\temp> & '.\Listing 10.1.ps1'
|09/06/2021 10:53:00|    |Start...............script|   |Information|
```

ADDUser2Group-Log_9-6-2021_10-53AM_.log - Notepad
File Edit Format View Help

```
|09/06/2021 10:53:00|    |Start...............script|   |Information|
|09/06/2021 10:53:00|    |Processing..........user1|   |Information|
|09/06/2021 10:53:00|    |ADD sukhijv to messaging offshore|   |Information|
|09/06/2021 10:53:00|    |Success - ADD user1 to messaging offshore|   |Information|
|09/06/2021 10:53:00|    |Processing..........user2|   |Information|
|09/06/2021 10:53:00|    |user2 is already member of bsc messaging offshore|   |Information|
|09/06/2021 10:53:00|    |Script Finished|   |Information|
```

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

In a similar manner, you can create numerous scripts tailored to various production needs. Over the past decade, I have contributed hundreds of scripts to the community, many of which are available for you to access and adapt to your specific requirements. You can find them at the following link: `https://techwizard.cloud/downloads/`.

These scripts are built upon the same core principles we've explored throughout this book. Feel free to modify them as necessary to suit your use cases, whether it's automating tasks, improving efficiency, or ensuring consistent workflows in your environment.

Product Examples (Daily Use)

In this section, I'm sharing some handy snippets that you can use as-is or incorporate into your own scripts for daily administrative tasks. Given my passion for Microsoft Exchange, I've focused on providing useful Exchange-related examples. These script excerpts are designed to simplify common Exchange management tasks, and you can easily integrate them into your automation routines.

Here are some Exchange script snippets you can use in your day-to-day operations.

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,Client
Hostname,ServerIp,ServerHostname,sender,EventId,MessageSubject,
TotalBytes , SourceContext,ConnectorId,Source, InternalMessageId,
MessageId ,@{Name="Recipients";Expression={$_.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 (AD) is the backbone of every Microsoft product, and with PowerShell, you can automate a variety of AD components to streamline management. Fortunately, Microsoft has developed a native Active Directory module specifically for this purpose.

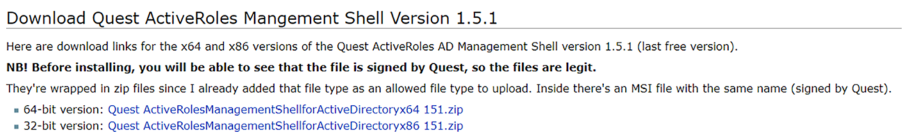
There are several methods available for Active Directory scripting through PowerShell, including

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

In the past, my personal favorite was the **Quest Management Shell** when it was freely available, but over time, Microsoft's **Active Directory**

Module has caught up—and, in my opinion, even surpassed it in terms of functionality.

One key reason the Microsoft AD module has become my go-to is that the **Quest AD module is no longer free**. While you can still find older versions online, such as version 1.5.1 (see Figure 10-4), it's important to check if any licensing is required before using it in a production environment. If you'd like to explore the older Quest AD module, you can download it here: Quest AD Management Shell v1.5.1.



Download Quest ActiveRoles Mangement Shell Version 1.5.1

Here are download links for the x64 and x86 versions of the Quest ActiveRoles AD Management Shell version 1.5.1 (last free version).

NB! Before installing, you will be able to see that the file is signed by Quest, so the files are legit.

They're wrapped in zip files since I already added that file type as an allowed file type to upload. Inside there's an MSI file with the same name (signed by Quest).

- 64-bit version: Quest ActiveRolesManagementShellforActiveDirectoryx64 151.zip
- 32-bit version: Quest ActiveRolesManagementShellforActiveDirectoryx86 151.zip

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

That said, I highly encourage you to use the **Microsoft Active Directory Module**. However, I understand that some admins and organizations continue to use the Quest AD module, either with the free version or a purchased license.

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 DisplayN
ame,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)

`LaunchSPO/RemoveSPO` (SharePoint online)

`LaunchCOL/RemoveCOL` (Security and Compliance)

`LaunchMSOL/RemoveMSOL` (MS Online Azure Active Directory)—This is being retired by Microsoft and they want user to use Microsoft graph module

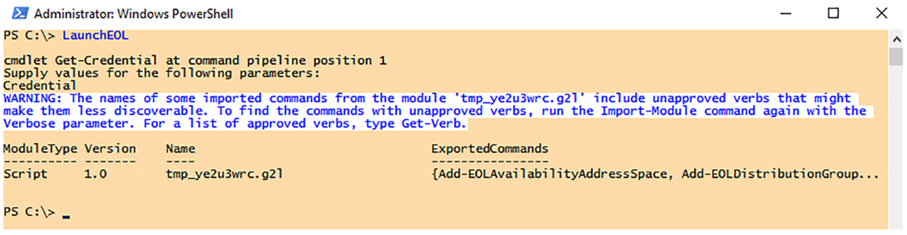
```
##################Exchange Modern Online##################
Function LaunchEOL {
  [CmdletBinding()]
  param
  (
    [Parameter(Mandatory = $false)]
    $Credential
  )
  Import-Module ExchangeOnlineManagement -Prefix "EOL"

Connect-ExchangeOnline -Prefix "EOL" -Credential
$Credential  -ShowBanner:$false
}
Function RemoveEOL {
  Disconnect-ExchangeOnline -Confirm:$false
}
############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:



```
PS C:\> LaunchEOL

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
WARNING: The names of some imported commands from the module 'tmp_ye2u3wrc.g2l' include unapproved verbs that might
make them less discoverable. To find the commands with unapproved verbs, run the Import-Module command again with the
Verbose parameter. For a list of approved verbs, type Get-Verb.

ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     1.0        tmp_ye2u3wrc.g2l                    {Add-EOLAvailabilityAddressSpace, Add-EOLDistributionGroup...

PS C:\> _
```

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,UserPrincipal
name,
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 "Exchange
ItemGroup","ExchangeItem","ExchangeAggregatedOperation" -Result
Size: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 (Entra)

While we've covered practical examples of managing traditional Active Directory, it's important to recognize that in today's landscape, **Azure Active Directory (Azure AD)** is becoming increasingly common. Below, I've provided some examples from the Azure AD world to help you get started.

To work with **Azure AD** in PowerShell, you'll need to use Connect-AzureAD to establish a connection. For **Microsoft Online Services (MS Online)**, you can use Connect-MsolService to connect.

Once you're connected, simply update the variables in the following examples to suit your environment, and you'll be ready to automate tasks within Azure AD.

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.

Microsoft Graph Module

As both the AzureAD and MSOL modules are becoming obsolete, Microsoft is now strongly encouraging everyone to transition to the Microsoft PowerShell Graph SDK. This shift is important to stay aligned with modern management practices, as the PowerShell Graph SDK provides a more comprehensive and future-proof way to manage Azure resources.

The Graph SDK offers a unified API for managing not only Azure AD but also a wide range of Microsoft services, from Office 365 to Microsoft Teams, SharePoint, and beyond. By adopting the Graph SDK, you'll gain access to an even broader set of tools and capabilities that go far beyond what AzureAD and MSOL modules offer.

Here are some key reasons to make the switch:

- **Unified Access:** With the Graph SDK, you can manage multiple Microsoft services from a single interface, eliminating the need to juggle separate modules for each service.

- **Regular Updates:** Microsoft is actively enhancing the Graph API, ensuring it remains the most up-to-date and secure way to interact with Microsoft services.
- **Broader Scope:** The Graph SDK enables you to automate workflows across Azure AD, Exchange, Teams, and many other services, providing a more integrated automation experience.

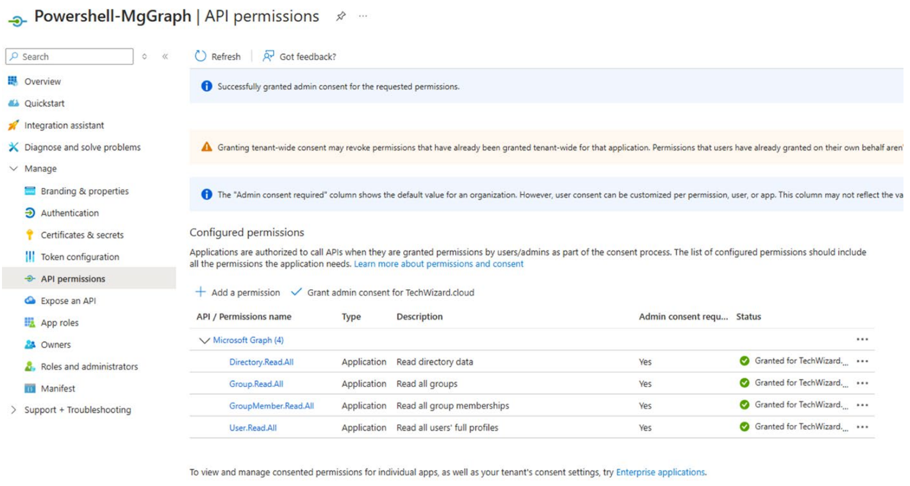
To get started with the Microsoft PowerShell Graph SDK, you will need to install the module via the following:

Install-Module Microsoft.Graph

Now you need to register the APP in Azure AD, ADD the required API permissions, for example, user.read.all, directory.read.all, etc.

Create a certificate and add the thumbprint to the Registered AzureAD application.

See Figure 10-6 showing azure AD app registration API permissions.



Successfully granted admin consent for the requested permissions.

Granting tenant-wide consent may revoke permissions that have already been granted tenant-wide for that application. Permissions that users have already granted on their own behalf aren

The "Admin consent required" column shows the default value for an organization. However, user consent can be customized per permission, user, or app. This column may not reflect the va

Configured permissions

Applications are authorized to call APIs when they are granted permissions by users/admins as part of the consent process. The list of configured permissions should include all the permissions the application needs. Learn more about permissions and consent

Add a permission Grant admin consent for TechWizard.cloud

API / Permissions name	Type	Description	Admin consent requ...	Status
Microsoft Graph (4)				
Directory.Read.All	Application	Read directory data	Yes	Granted for TechWizard....
Group.Read.All	Application	Read all groups	Yes	Granted for TechWizard....
GroupMember.Read.All	Application	Read all group memberships	Yes	Granted for TechWizard....
User.Read.All	Application	Read all users' full profiles	Yes	Granted for TechWizard....

To view and manage consented permissions for individual apps, as well as your tenant's consent settings, try Enterprise applications.

Figure 10-6.

Connect to Microsoft Graph:

Connect-MgGraph -ClientId $MgGClientID -CertificateThumbprint $ThumbPrint -TenantId $TenantName

Disconnect from Microsoft Graph:

Disconnect-MgGraph

Get user properties:

Get-MgUser -userid svikas@techwizard.cloud | Format-List ID, DisplayName, Mail, UserPrincipalName

Get all users:

Get-MgUser -All | Format-List ID, DisplayName, Mail, UserPrincipalName

AWS PowerShell Module

When it comes to managing and automating tasks in Amazon Web Services (AWS), PowerShell is a highly versatile and efficient tool that provides seamless integration with AWS's powerful suite of services. By combining the capabilities of AWS CLI (command-line interface) with PowerShell, users can unlock a wide range of functionalities that simplify operations, automate processes, and offer greater control over AWS infrastructure.

Run the following command in an elevated PowerShell session to install the module:

Install-Module -Name AWSPowerShell

Configure AWS credentials:

Set-AWSCredentials -AccessKey $AccessKey -SecretKey $SecretKey

Now you can utilize **Get-Ec2Instance, Get-Ec2Volume**, and hundreds of other commands.

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

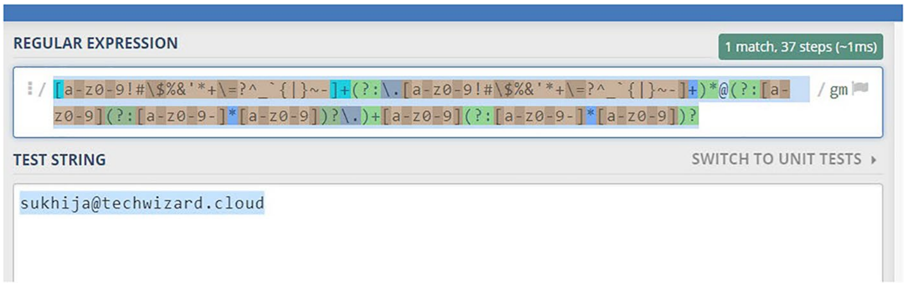
```
$filep = "c:\file.txt"
$getNetworkID = Get-Content $filep | where { $_ -ne "" }
@("Employeeid") +  $getNetworkID | Set-Content  $filep -Force
#add emplyeeidheader
```

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.

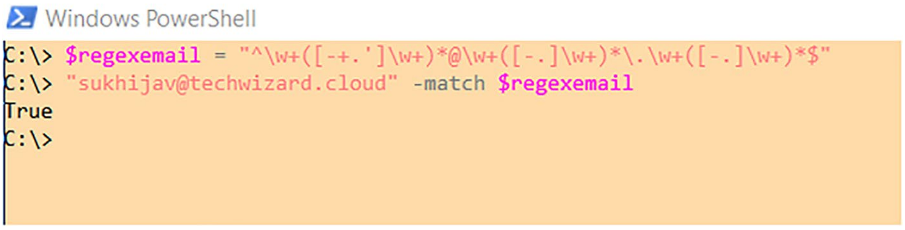
This is how you use it in PowerShell, and it will be used mainly with match operators. See Figures 10-7 and 10-8.



```
/ [a-z0-9!#\$%&'*+\=?^_`{|}~-]+(?:\.[a-z0-9!#\$%&'*+\=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])? / gm
```

```
sukhija@techwizard.cloud
```

Figure 10-7. *Showing regular expression testing*



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

Figure 10-8. *Showing a regular expression operation in PowerShell*

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

S. no.	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 final chapter has been all about showing you how to combine different snippets of PowerShell code to create powerful scripts that can handle bulk administrative tasks efficiently. Throughout this book, I've shared practical examples from various products that system administrators can use in their daily operations to automate repetitive work, streamline workflows, and increase productivity.

For even more scripts and hundreds of additional examples, feel free to visit `https://techwizard.cloud/downloads/`, where I've provided a vast collection of ready-to-use solutions that you can customize to suit your environment.