PowerShell Succinctly

by Rui Machado

Syncfusion | Technology Resource Portal

PowerShell Succinctly

By
Rui Machado

Foreword by Daniel Jebaraj

Copyright © 2014 by Syncfusion Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

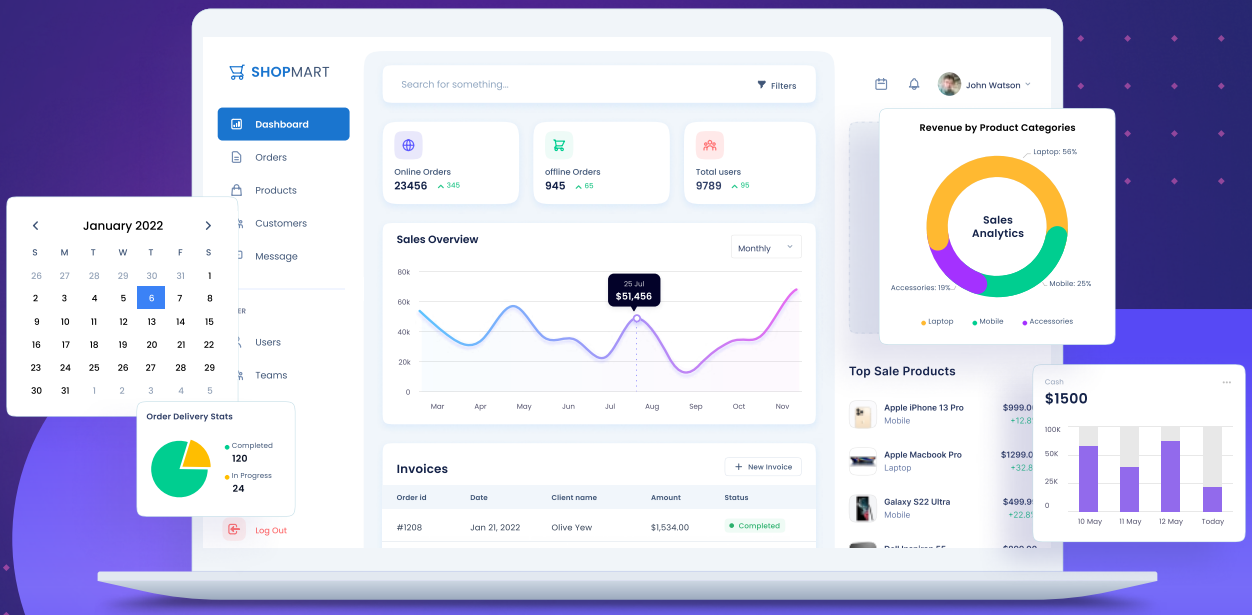
Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Jeff Boenig
Copy Editor: Ben Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING

POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

4.6 out of 5 stars

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Table of Contents

The Story behind the *Succinctly* Series of Books **8**

About the Author **10**

A special thanks 10

Introduction **11**

How PowerShell is different 11

Who is this book for? 12

Code Samples 12

Notes 12

Chapter 1 Basics **13**

PowerShell Version 13

PowerShell Interactive Shell 13

The Shell and Existing Tools 13

Execution Policies 15

How to Run a Program 17

How to Run a Script 17

How to Run Commands 18

Get Help with Existing Commands 18

Read and Write from the Interactive Shell 18

PowerShell Snap-Ins 19

Add a Snap-In to a Script 19

Add a Script Reference to Another Script 20

Using .NET classes 20

Pipelines 21

Variables 22

Get properties from an Item 23
Format Variable Output 24
Strings 25
PowerShell Here Strings 26
Regular Expressions 27
Lists of Items 28
Arrays 28
.NET Lists 28
Hash Tables 29
Flow Control 29
Logical and Comparison Operators 29
Conditional Statements 32
Loops 33
Managing the Flow 35
Schedule Script Execution 35
Extensibility and Code Reuse 38
Create instances of objects 38
Functions and Parameters 38
Create a Windows Form 43
Chapter 2 File System 47
Current Location 47
Get Files from a Directory 47
Get the Content of a File 48
Manipulate the Content of a File 49
Create Temporary Files 50
Manage Directories 50
Create New Directories 50

Change Directory Permissions 51
Remove Directories 52
Rename Directories 52
Move a File or a Directory 53
Managing Paths 53
Join Parts into a Single Path 53
Split Paths into multiple parts 54
Test if Path Exists 55
Resolve Paths 55
Chapter 3 Processes 57
List All Processes 57
Get a Process by ID 58
Stop a Process 58
Start a Process 58
Chapter 4 Windows Management Instrumentation 60
Using WMI classes 60
Access WMI Classes 61
Exercise: Get Available Disk Space 62
Chapter 5 Remote PowerShell 65
Using Remote PowerShell 65
Identify Remote PowerShell Compatible Commands 68
Test a Remote Connection 68
Invoke Scripts in Remote Machines 69
Chapter 6 Structured Files 71
Manipulating XML Files 71
Import XML from File 71
Load XML File from String 71

Export XML to File 72
Manipulating CSV Files 74
Import CSV from File 74
Export CSV to File 75
Load CSV and Send Email 76
Manipulating TXT Files 77
Import TXT from File 77
Export TXT to File 78
Using XSL to Transform XML Files 78
Chapter 7 SQL Server and PowerShell 81
Install SQLPS 81
Add SQL Snap-in 82
Invoke SQL Query 82
Chapter 8 Microsoft Office Interop 85
Using PIAs Assemblies 85
Create an Instance of an Excel Application 86
Retrieve Data from Excel File 87
Exercise: How Many SQL Server Connections are in That Excel File? 89
References 91

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese.”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Rui Machado is a developer and a software engineer, born in Portugal, with a strong .NET background and a special interest in scripting languages. He holds a graduation in Information Technologies and a post-graduation in business intelligence (BI) and is now finishing his master's degree in Information Systems. He works for ALERT Life Sciences, a Portuguese software house for clinical solutions, as a business intelligence engineer.

Since his first VB.NET application in 2007, Machado has worked with C#, PowerShell, BizTalk, Integration Services, and Analysis Services, making him a technology enthusiast with a lot of love for programming languages. He is now focused on the Oracle Data Integrator and microstrategy for BI solutions.

PowerShell entered his professional life while Machado was developing a BizTalk integration project in which the client had no licenses for use of that technology and didn't want to pay for a single integration project. The solution was to use a scripting language to make several systems connect, apply transformations, and build an efficient data flow using PowerShell. Since that first PowerShell project he has started a PowerShell blog for the Portuguese community, written in several forums, and been a speaker in several events.

Although most of his time is spent working for and with technology, Machado also manages to have time for other activities like skateboarding, surfing, and enjoying life with friends and family. After all, life won't give you its best if you don't give your best to it.

A special thanks

There were several people throughout my career that aided me in becoming the professional that I am today. Besides allowing me to develop my technology passion, these people gave me their knowledge because they believed I could use it correctly. PowerShell is one of those cases; this scripting technology wouldn´t be a part of my career if it wasn´t for my coworker José Antonio Silva, who spent several hours teaching me how to take my first steps in this technology, and Sandro Pereira, who taught me that sharing knowledge is not a loss but a gain for all our community members. Thanks also to everyone that works for the evolution of technology, and to all my family and friends that continue believing in me and what I work for.

Introduction

Windows PowerShell might be a well-known scripting language for some system administrators, who see it as an evolution of the former Windows command-line shell and use it for their daily systems management activities, but that isn't the prevailing opinion.

PowerShell brought several new concepts like object-oriented pipelines, which revolutionized the way users invoke commands and create scripts, but most developers and systems administrators fear this language and continue to avoid it. Although PowerShell is different for many developers, you will see in this book that it offers a number of advantages.

How PowerShell is different

PowerShell opens up a new world of opportunities compared to other command-line shell scripts, starting with the way the shell interprets the commands you use. In traditional command-line shells, commands are interpreted as plain text and don't allow any kind of interaction. PowerShell interprets objects with methods and attributes that have access to much of the .NET framework, resulting in a more powerful programming model.

One simple example of this feature can be demonstrated with how PowerShell allows you to retrieve the length of an object, shown in the following code sample:

```
"Rui".Length
```

This means that when PowerShell encounters a quoted string, it will automatically build an object of type `System.String`, allowing you to invoke any available method, attribute, or property of this .NET object type. In this case, I used the `length` property to give me the length of this string.

As you can see in the previous example, PowerShell uses the .NET Framework, but it's not the only technology that PowerShell integrates in its shell. It allows you to work with .NET, but also COM, WMI, XML, and Active Directory, enriching the power of this scripting language.

The last difference I want to note is the type of command used by PowerShell. They are no longer text based. Instead, they use a new type called *cmdlets*. These *cmdlets* have a proper syntax "verb-noun" and are task based, which means that just by looking at its name you can guess what task it will perform. For example, if you want to see all your active processes, you can invoke the following *cmdlet*.

```
Get-Process
```

Who is this book for?

This book is being written primarily for system administrators and .NET developers. Although PowerShell is mainly used by systems administrators, I want to show .NET developers how they can use this scripting language for several daily activities, like testing their software.

What developers often do when they want to test something like a WCF web service is build a console application with their client call code, compile it, and run. With PowerShell, you can forget about opening Visual Studio, selecting the creation of a new console application, and so on. Just open your script editor, write your code, and run. You will save a lot of time, I assure you. So this book is not just for those who use scripting languages to manage machines. I hope that by the end of this book, everyone who uses a computer every day will realize they can learn PowerShell and use it to automate some of their activities.

Code Samples

All of the examples in this book were created in PowerGui script editor, but are perfectly compatible with the integrated Windows editor, PowerShell ISE, and the PowerShell Console.

In this book, the code samples are shown in code blocks such as the following example.

```
function GetProcess($id)
{
      Get-Process -Id $id
}

GetProcess -id 13108
```

Notes

There are notes that highlight particularly interesting information about a language feature, including potential pitfalls you will want to avoid.

Note: An interesting note about PowerShell.

Chapter 1 Basics

PowerShell Version

PowerShell is now in version 4.0 and this book has been written in version 3.0. This means that some features will not be available for you if you are using a version under 3.0. To check your PowerShell version, run the command `Get-Host` and then look for the version property. If you have any version under 3.0, please download it in the this link, which will install the Windows Management Framework 3.0 in which PowerShell is included (click here). Be careful, as it is only available for Windows 7 or higher, and for Windows Server 2008 or higher.

PowerShell Interactive Shell

The Shell and Existing Tools

To start using the interactive shell of PowerShell, you just need to run PowerShell.exe instead of the more commonly used shell cmd.exe. If you have a 64-bit system, you will have two versions of PowerShell to choose from, a 32-bit version and a 64-bit version.

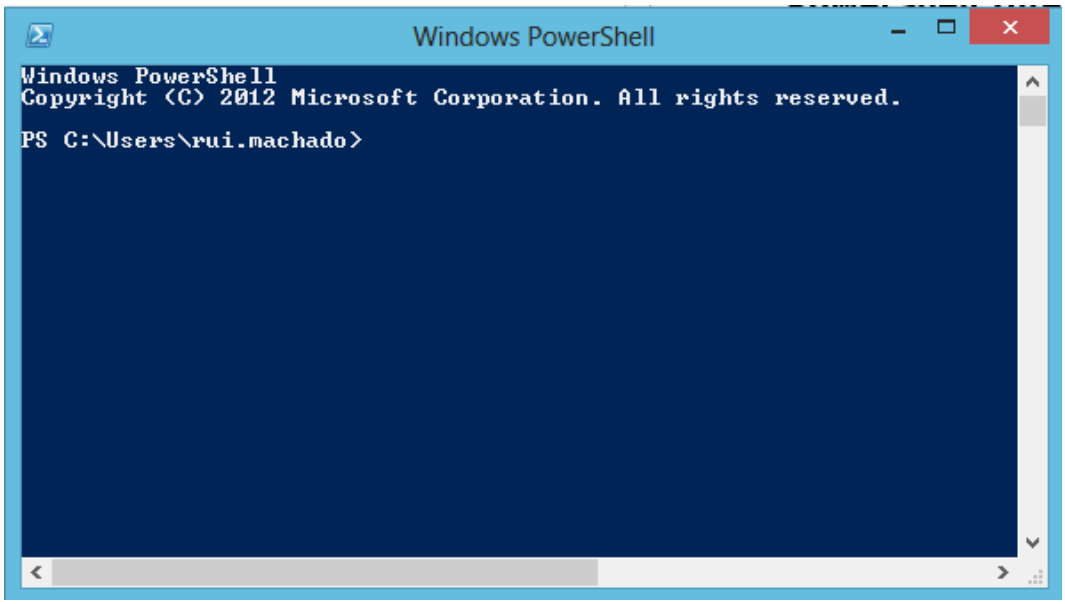


Figure 1: Interactive Shell

Once you open this interactive shell, you can start typing PowerShell commands (*cmdlets*) and getting results. A nice feature of this PowerShell command prompt is that it is compatible with the former DOS commands and UNIX commands. This might be useful if you are accustomed to navigating between directories and files with this kind of syntax.

```
PS C:\Users> ls

    Directory: C:\Users

Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d-----        17/04/2013     15:18            .NET v2.0
d-----        17/04/2013     15:18            .NET v2.0 Classic
d-----        17/04/2013     15:19            .NET v4.5
d-----        17/04/2013     15:18            .NET v4.5 Classic
d-----        17/04/2013     15:18            Classic .NET AppPool
d-----        18/04/2013     14:09            DefaultAppPool
d-----        16/04/2013     15:54            MsDtsServer110
d-----        16/04/2013     16:04            MSOLAP$PRI01
d-----        16/04/2013     16:03            MSSQL$PRI01
d-----        16/04/2013     16:03            MSSQLFDLauncher$PRI01
d-r---        04/04/2013     11:30            Public
d-----        16/04/2013     16:04            ReportServer$PRI01
d-----        09/06/2013     21:29            rui.machado
d-----        10/05/2013     10:15            SQLAgent$PRI01
```

Figure 2: Executing Scripts with UNIX and DOS syntax

This interactive shell is all you need to run simple commands; however, when it comes to automating tasks and managing scripts, this shell can become insufficient and it is useful to start using a script editor. You can use Notepad if you like, but there are several tools that provide syntax highlighting and intelligent editing features along with script debugging and an integrated shell for compiling. Along with your interactive shell, you also have, out of the box, a nice script editor from Microsoft, PowerShell ISE, which provides better script management and a full list of the available PowerShell commands. This tool is integrated with Windows and it's applied for PowerShell versions 2.0, 3.0, and 4.0.

To open the PowerShell ISE tool, click **Start**, select **Run**, and then execute the following command: "powershell_ise.exe". This will open a new PowerShell ISE instance, which looks like the following figure:

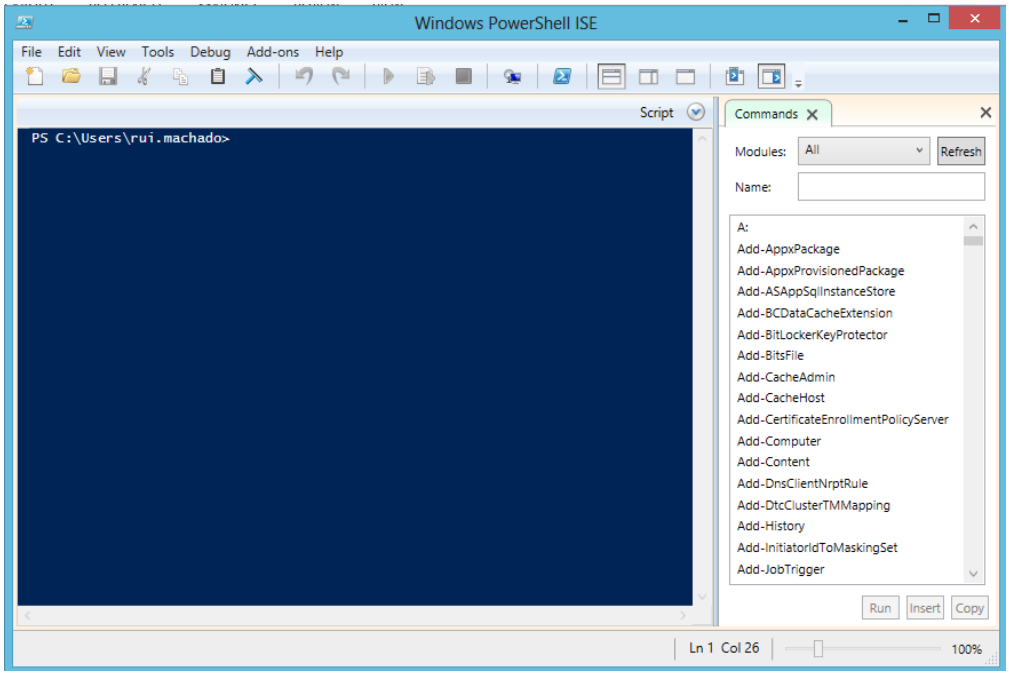


Figure 3: PowerShell ISE

Although it's better that the simple interactive shell, PowerShell ISE might not be enough for you, as it still uses the command prompt to write scripts. It also doesn't have different colors for methods, attributes, and commands, what I call *Color Sense*, or more often described by many IDEs as syntax highlighting. To optimize your productivity with PowerShell even more, you might think about using PowerGui, a free script editor often used for development.

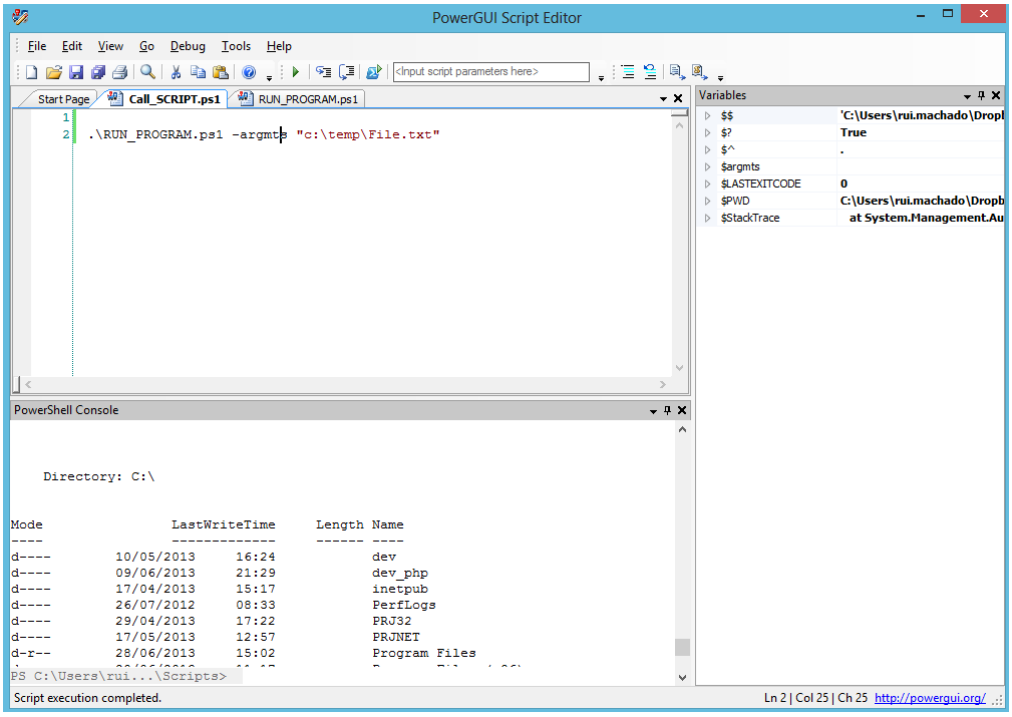


Figure 4: PowerGui script editor

This script editor provides you with a list of variables in use and its last value for debugging, a PowerShell console for you to check the result of your commands, and many more features you can try later. If you are a daily developer of PowerShell, I recommend you use a script editor like this one. You can download PowerGui here.

Execution Policies

PowerShell has a very unique security protocol, referred to as the execution policy, which allows you to define the type of scripts that can run on your machine or workgroup. As I mentioned previously, choosing one of the five available execution policies will determine if the execution of all scripts is allowed, only execution of scripts typed in the interactive shell is allowed, or if permission to execute scripts is based on a rule. If you want to run scripts in an external script editor like PowerGui, you must change your execution policy or you will receive the error shown in Figure 5.

```
. : File C:\Users\rui.machado\AppData\Local\Temp\9be7b462-c5f7-4175-b534-2dc42d043a8e.ps1 cannot be loaded
because running scripts is disabled on this system. For
more information, see about_Execution_Policies at http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . 'C:\Users\rui.machado\AppData\Local\Temp\9be7b462-c5f7-4175-b534-2dc42d043a8e. ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

Figure 5: Error running scripts without permission

By default, PowerShell will only allow you to execute scripts through its interactive shell, which prevents external script execution. However, by setting the maximum level of security, you can easily set other levels of security. The full list of execution policies are shown in the following table.

Table 1: Execution Policies

Execution Policy	Level of Security
Restricted	Will only allow interactive shell execution.
AllSigned	Runs only scripts with a digital signature. Executing a script for the first time will prompt a trust publisher message.
RemoteSigned	All scripts from the Internet must be signed.
Unrestricted	Allows any script execution. Scripts from the Internet must be trusted.
ByPass	Every script execution must be validated by the user.

To change your execution policy, you must use the interactive shell. To open the interactive shell, click **Start** and select **Accessories**. Run **Windows PowerShell** and make sure you start it with elevated privileges (**Run as Administrator**). Type the command to change it to the one you want. To do so, right-click the Windows PowerShell icon in the Accessories tab and then click **Run as Administrator**.

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

```
Windows PowerShell
Copyright (C) 2012 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> Set-ExecutionPolicy -ExecutionPolicy Restricted

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose
you to the security risks described in the about_Execution_Policies help topic at
http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Windows\system32>
```

Figure 6: Change Execution Policy

To see the execution policy currently activated in your system, you can use the `Get-ExecutionPolicy` command. You should now see **RemoteSigned** as your current execution policy.

How to Run a Program

PowerShell allows you to start a program from its own shell, so that you won´t waste all your existing executables like Perl scripts or a simple batch file. To start that program you want, type in PowerShell its name followed by any arguments you might need or want. If the program name has spaces, enclose its name in quotation marks after you type an ampersand and followed by its arguments. An example of this is shown in the following code sample.

```
$args = ".\My File.txt"

& 'C:\Windows\System32\notepad.exe' $args
```

Executing this command on my computer will open a text file I have already created, as you can see in the following figure.

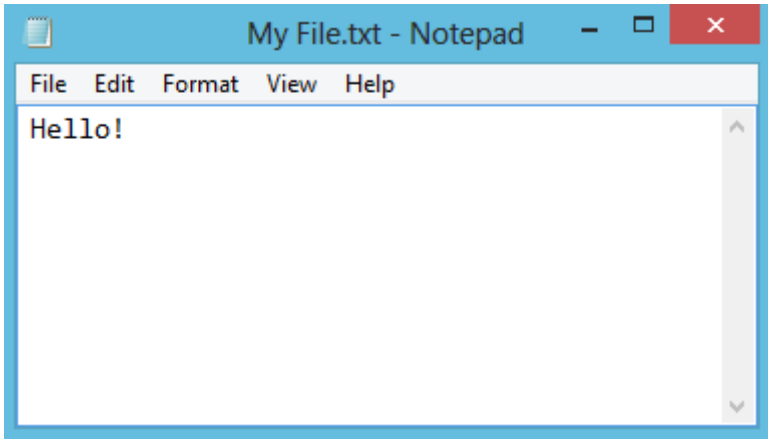


Figure 7: Run a Program

How to Run a Script

If the execution policy you have set up allows you to execute external scripts, type the name of your script in the command prompt, just like any *cmdlet*.

```
.\RUN_PROGRAM.ps1
```

If your script has parameters, you can also pass them directly from the command prompt.

```
.\RUN_PROGRAM.ps1 -argmts "c:\temp\File.txt"
```

How to Run Commands

Running commands is as easy as typing them in the interactive shell command prompt or any other script editor and the result will be shown to the user. An example of this scenario can be invoking the command to retrieve all script execution history.

```
Get-History
```

```
PS C:\Windows\system32> Get-History

  Id CommandLine
  -- -----------
   1 Set-ExecutionPolicy -ExecutionPolicy Restricted
   2 Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
   3 Get-Jobs
   4 Get-Job
   5 Get-Item
   6 Get-Host
```

Figure 8: Run a Command (Cmdlet)

Clean the shell

Whether programming in the Interactive shell or in a Script Editor with the execution option, to clean your result use the `Clear` or `CLS` command.

Get Help with Existing Commands

To get information about a specific command, you can use the `Get-Help` command, followed by the command name.

```
Get-Help -Name Get-History
```

When you run this command for the first time, PowerShell might ask you to download the most recent help library, as shown in Figure 9, to provide you with the newest information about its commands.

```
PS C:\Windows\system32> Get-Help

Do you want to run Update-Help?
The Update-Help cmdlet downloads the newest Help files for Windows PowerShell modules and installs them on your
computer. For more details, see the help topic at http://go.microsoft.com/fwlink/?LinkId=210614.
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

Figure 9: Update Help

Read and Write from the Interactive Shell

To write a value to the shell, you can use the `Write-Host` command. This not only writes a value to the console, but also allows you to customize its output, such as the color of text by using the ForegroundColor parameter, or the background color by using the BackgroundColor parameter.

```
Write-Host -ForegroundColor White "SyncFusion" -BackgroundColor Black
```

Invoking the previous command will result in the following display in the interactive shell.

Figure 10: Write-Output Customization

Tip: When PowerShell finds a string that is not captured by a context, it will print its value. If you want to write a variable, just write its name in the console.

```
PS C:\Users\rui.machado> $string="SyncFusion"
PS C:\Users\rui.machado> $string
SyncFusion
```

Figure 11: Context Printing

To read a value from the interactive shell, you can use the `Read-Host`, which reads a line of input from the console.

```
$value = Read-Host
```

PowerShell Snap-Ins

A PowerShell snap-in is a way of extending the commands available in the shell with your own or some downloaded from the Internet. These snap-ins originated from a C# block of code, an implementation of a PSSnapin class to be more precise, compiled and imported to a script with its DLL (dynamic-link library). This might be useful if you have a set of methods in an already finished helper class that you want to reuse to save time developing those methods again. You can´t use this DLL directly, as you need to add some metadata to your methods so that PowerShell can identify a *cmdlet*, but it's a huge advantage for developers.

Add a Snap-In to a Script

To add a snap-in to your script, you can invoke the following code:

```
Add-PSSnapin -Name "MySnap-in"
```

You will only need to add the snap-in to one of your scripts to use it in your session; while your PowerShell Scripts are running, that snap-in is available.

Note: You can only add to your session registered snap-ins. To check the full list of registered snap-ins, run "Get-PSSnapin –registered". If you have downloaded or created a new snap-in, you can register it with the InstallUtil tool included with the Microsoft .NET Framework.

To register a snap-in, launch the PowerShell interactive shell with elevated privileges and do the following:

1. Change the directory to C:\Windows\Microsoft.NET\Framework\v4.0.30319.
2. Run InstallUtil.exe “MySnapin.dll”.

```
PS C:\Users\rui.machado> cd C:\Windows\Microsoft.NET\Framework\v4.0.30319
PS C:\Windows\Microsoft.NET\Framework\v4.0.30319> InstallUtil.exe "MySnapin.dll"
```

Figure 12: Register a New Snap-In

Add a Script Reference to Another Script

Referencing scripts is a useful task to reuse existent code blocks like your own functions, which will allow you to save time making the same scripts several times. This way you will only develop once and can use it several times. To add a script reference to another script, you just need to use the command `Import-Module`.

```
Import-Module "MyOtherScript.ps1"
```

Note: All of these commands have optional parameters to fulfill special needs. This way you can always use the command Get-Help “Command Name“ to see the full parameter list.

Using .NET classes

To use a .NET class in PowerShell, you just need to identify the namespace between brackets and then use a double colon to call the method you want. Parameters are passed the same way in .NET. For example, you can see what today’s date and time are by invoking `[DateTime]::Now`.

Pipelines

Pipelines are an important and heavily used feature in PowerShell. A concept borrowed from UNIX, pipelines let you use the result of a command execution as input to another command. This way you don´t need to save the result in variables, but instead just create a flow of data in a logic block of code. In PowerShell, the pipe character is "|" and you can use it with any command.

Where and Select

Where-Object can also be used with the alias "where" or the symbol "?"

Select-Object can also be used with the alias "select"

A key feature about PowerShell pipelines is that they don´t pass the result of a command as text, but rather as an object, allowing you to invoke other complementary commands on a particular result set. By using commands like `Where-Object` to filter the set or `Select-Object` to select specific properties of an object, pipelines can be an extremely advanced concept to query result sets of commands.

In the previous code block, there is a simple example. I used the command `Get-Process` to retrieve all active processes and then filtered the result list to get only the processes that have the name "PowerShell" with the command `Where`. I then selected only the property ID from the object with the command `Select`.

Another important concept surrounding pipelines is the current pipeline element, accessible by the command $_. This special variable is used to reference the current value in the pipeline. In this case, $_ is a collection of rows produced by the Get-Process command, in which each row contains a Name property. The `where` command filters the rows by comparing the Name property to "PowerShell" and sends the resulting rows to the next stage of the pipeline.

```
Get-Process | where{$_.Name -like "PowerShell"} | Select Id
```

While using pipelines, you might need to break lines to make a more readable flow. To do that, you need to use the PowerShell break line special character (line continuation character) which is the back quote "`" for breaking strings or the pipeline "|" character to break after a pipe element.

```
Get-Process | `
      where{$_.Name -like "powershell"} | `
                Select Id
```

Note: Be careful with the usage of several pipelines, the resulting flow might become difficult to understand for developers and system administrators.

Variables

Variables are used in every programming language to store information that you wish to manipulate later in your program or script structure. The same is true in PowerShell; variables are used to store information about a command result to use it later or to pass it to a different pipeline level.

In PowerShell, variables start with the character $ (dollar symbol) and are followed by almost any character. The only restrictions are the PowerShell special characters. PowerShell only releases the contents of variables when the user session ends, which means that if you don't need to use a variable anymore, clean it by assigning the $null variable to it.

```
$var1=$null
```

Variable object type is resolved dynamically, which means that if you assign a string type object, the variable object type will be string, if it is XML then it will be XML. However, you can cast a variable to a different type using the type between brackets ([]) before the variable name. This can be useful when you don´t want to create a new instance of a .NET object; instead you can just cast it to the expected type and PowerShell will resolve it.

```
$var1 = "PowerShell"

$var2 = $var1

#Casting to DateTime
[System.DateTime]$var3="2013-06-13"

#Casting to Xml
$var4 = [System.Xml.XmlDocument]"<xml><node>HERE</node></xml>"
```

Variables in PowerShell can be defined with different scopes, so that you can set either a variable to be accessible only from a specific script, only within a session, or make it accessible to the entire shell. The variables full scope list possibilities are shown in the following table.

The default scope of a variable differs according to the place in which you define that variable. If it is defined in the interactive shell it is Global, if outside any function or script block it is Script and otherwise it is local.

Table 2: Variable Scopes

Scope Name	Scope
Script	Only available in that script.
Local	Available only on a specific scope or sub-scopes, like a loop and any loops inside.
Private	Available only on a specific scope, like a loop inside.

Scope Name	Scope
Global	Available in the entire shell.

Get properties from an Item

Listing the properties of an item is one of the main activities for system administrators so that they can make decisions based on a particular object property value. PowerShell provides you with a command to list detailed information about a particular object. To show that list, use the command `Format-List` on any object.

```
Get-Process | `
      where{$_.Name -like "powershell"} | `
                Format-List
```

This property list can also be stored in variables for any future usage.

```
$var1 = Get-Process | `
      where{$_.Name -like "powershell"} | `
                Format-List
```

PowerShell allows you not only to save the result of that command, but also to store the command itself by enclosing them in curve brackets “{}”.

```
$var1 = {Get-Process | `
      where{$_.Name -like "powershell"} | `
                Format-List}
```

The difference between these two calls is shown in the following figure.

```
#With Curve Brackets

Get-Process | `
    where{$_.Name -like "powershell"} | `
        Format-List

#Without Curve Brackets


Id      : 3116
Handles : 288
CPU     : 0,5
Name    : powershell
```

Figure 13: Using Curve Brackets in Commands

Format Variable Output

There are three commands to format the output of a variable. You can use `Format-List` to format your output as a vertical list, `Format-Table` to output your result as a table and `Format-Wide` to retrieve single-data item and show it as multiple columns.

```
Get-Process | Format-List
Get-Process | Format-Table
Get-Process | Format-Wide

#Choose how many columns you want
Get-Process | Format-Wide -Column 3
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    348      37   130064     146500   325    44,64  11456 AcroRd32
    244      17     7708      15344    96     0,36  12504 AcroRd32
     75       7     1124       3856    44           1960 armsvc
   2673     222   336544      77760   716           4764 avp
   1210      48    10480      14180   128     5,02  10232 avp
     55      11     1120       3908    29           6492 BTHSAmpPalService
    228      26     3712       8380    46            872 BTHSSecurityMgr
    105      10     1620       5652    63           1336 BtwRSupportService
    152      21    21392      23748   172     0,38   956 chrome
    174      40    52688      62608   219     8,09  1268 chrome
    158      35    36396      45564   191     7,03  4992 chrome
    156      33    33132      40500   191     1,84  5344 chrome
    156      32    31600      38596   190     2,25  6944 chrome
    159      35    42356      52132   195     6,30  7024 chrome
    157      30    31340      40932   191     1,81  7892 chrome
    170      28    58912      67364   215     5,61  8244 chrome
   2348      94   147772     165992   442   181,83  8568 chrome
```

Figure 14: Format-Table

```
Id      : 10256
Handles : 158
CPU     :
Name    : winvnc

Id      : 976
Handles : 354
CPU     : 5,984375
Name    : WINWORD

Id      : 7688
Handles : 1150
CPU     : 2015,6875
Name    : WINWORD
```

Figure 15: Format-List

```
AcroRd32                AcroRd32
armsvc                  avp
avp                     BTHSAmpPalService
BTHSSecurityMgr         BtwRSupportService
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
chrome                  chrome
```

Figure 16: Format-Wide with two columns

Strings

Strings are one of the most commonly used variable types in programming languages and PowerShell is no exception. Strings are used to compare usernames, to evaluate an OS version, and even to compare XML nodes. By default, strings in PowerShell are .NET System.String objects, which means that when you initialize a variable with a string, PowerShell will instantiate a System.String object with all its properties and methods.

```
$myVar = "PowerShell"
($myVar -is [System.String])
```

```
PS C:\Users\rui.machado> $string = "Rui"
PS C:\Users\rui.machado> $string -is [System.String]
True
```

Figure 17: Evaluate System.String type

Although using strings is simple in PowerShell, it introduces a new concept that usually confuses developers, which is literal and expanding strings. In literal strings, enclosure between single quotes (‘) and all text between commas are part of the string. When you enclose text with double quotes (“), PowerShell replaces all variable names and special characters with their values.

```
$a="SyncFusion"

#Literal strings
$literal = '$a`n and PowerShell'

#Expanding strings
$expanding = "$a`n and PowerShell"
```

The result of the previous code block is shown in Figure 18.

```
PS C:\Users\rui.machado> $a="SyncFusion"
PS C:\Users\rui.machado> $literal = '$a`n and PowerShell'
PS C:\Users\rui.machado> $expanding = "$a`n and PowerShell"
PS C:\Users\rui.machado> $literal
$a`n and PowerShell
PS C:\Users\rui.machado> $expanding
SyncFusion
 and PowerShell
```

Figure 18: Literal vs Expanding Strings

PowerShell Here Strings

PowerShell introduces a nice feature that simplifies the way you manage large text strings. This allows you to define multi-line strings without having to concatenate several string parts. To use a here string, you start by typing the special character "@" followed by a new line, and end your string with a new "@" character.

```
#simple string
$simpleString="Example String"

#Empty here string
@"

"@

#PowerShell here string
$largeString= @"
      Hello, this
      is a PowerShell here string. You can define
      multiple string lines into one single text block.
      You can even use special characters like `n or expand it with another
string
      $simpleString
"@
```

If you run this script in the interactive shell, you will see that PowerShell interprets this here string as an expanding multiline string. This allows you not only to simplify the way you initialize a multi-line string, but also the way you build the expanding stringNote that here strings can be either expanding or literal strings.

```
PS C:\Users\rui.machado\Documents> #PowerShell here string
PS C:\Users\rui.machado\Documents> $largeString= @"
>> Hello, this
>> is a PowerShell here string. You can define
>> multiple string lines into one single text block.
>> You can even use speacial characters like `n or expand it with another string
>> $simpleString
>> "@
>>
PS C:\Users\rui.machado\Documents> $largeString
Hello, this
is a PowerShell here string. You can define
multiple string lines into one single text block.
You can even use speacial characters like
 or expand it with another string
Example String
```

Figure 19: PowerShell Here Strings

Regular Expressions

Regular expressions are an easy and very efficient way of searching for patterns of characters in strings. While developing PowerShell scripts, you will need to use regular expressions in several contexts, which may include replacing text inside strings, retrieving all elements inside an XML document that contain a certain text or grouping blocks of text depending on a certain pattern. To use regular expressions in PowerShell, you can use the –match operator if you just want to evaluate a pattern inside a string or the –replace operator if you want to replace characters inside a string based on a given pattern. The following code block shows you how to use the –match operator.

```
#Text to evaluate.
$text = @"
      This is a PowerShell string!
      We will use it to test regular expressions
"@

#Evaluate a pattern. Returns:True
$text -match "\w"

#Evaluate a pattern. Returns:False
$text -match "!\w+"
```

Although evaluating a pattern in a string is a common task in PowerShell, replacing text is also important to make replacements based on a pattern's occurrence. The following code block shows you how to use the –replace operator. If you don´t understand regular expressions and don´t understand the meaning of "\w" or "\w+!", you can learn more on this in the MSDN webpage Regular Expression Language - Quick Reference. You will find a full list of characters you can use inside regular expressions and their purpose.

```
#Text to evaluate.
$text = @"
      This is a PowerShell string!
      We will use it to test regular expressions
"@

#Replace every character with the A character.
$text -replace "\w","A"

#Replace every string followed by !, with the B character.
$text -replace "\w+!","B"
```

Invoking the previous code block will result in the following string.

```
PS C:\Users\rui.machado\Documents> #Replace every character with the A characters
PS C:\Users\rui.machado\Documents> $text -replace "\w","A"
AAAA AA A AAAAAAAAAA AAAAAA!
AA AAAA AAA AA AA AAAA AAAAAAA AAAAAAAAAA
PS C:\Users\rui.machado\Documents>
PS C:\Users\rui.machado\Documents> #Replace every string followed by !, with the B character
PS C:\Users\rui.machado\Documents> $text -replace "\w+!","B"
This is a PowerShell B
We will use it to test regular expressions
```

Figure 20: Using -replace operator

Lists of Items

Arrays

To create an array in PowerShell, you can use one of two options: declare it explicitly by enclosing all your values inside @() and separated by commas, or implicitly by just separating them with commas as you can see in the following code block:

```
#Explicitly
$array1 = @(1,2,3,4,5,6,7,8,9,10)
#Implicitly
$array2 = 1,2,3,4,5,6,7,8,9,10
#Accessing with a literal index
$array1[2]
#Accessing with a variable index
$index=4
$array2[$index]
```

To access a value of the array, just invoke the variable and the index you want in square brackets i.e. `$array1[$index]`.

.NET Lists

Since you can use .NET classes and objects, you are able to use any kind of list from Collections.Generic collections. An example of using a list of strings is shown in the code block below.

```
$myList = New-Object Collections.Generic.List[string]

$myList.Add("Rui")

$myList.Item(0)
```

Hash Tables

Hash Tables in PowerShell are declared using the @ character and your Key=Value items enclosure in curly braces and separated by semicolons.

```
$values= @{
    "KEY1" = "VAL1"
  ; "KEY2" = "VAL2"
  ; "KEY3" = "VAL3"
}
```

To access a value of your hash table, just invoke the variable with the key of the entry in square brackets i.e. `$values["KEY1"]`.

Tip: Separate your values with semicolons at the start of each entry; it is easier to exclude one by commenting it at the start of its declaration.

Flow Control

Logical and Comparison Operators

To make decisions along your PowerShell scripts, you need to use conditional statements like in any other programming language. These conditional statements allow you to interact with data flowing in your script to define behaviors by its values.

All PowerShell operators start with a hyphen (“-“) followed by the desired operator name.

Note: PowerShell default operators are case insensitive; to use case sensitive operators, you need to prefix the operator with the “-c“ parameter.

In the following table, you can check all available PowerShell operators.

Table 3: Comparison Operators

Operator	Definition
-eq	Equals. Allows you to compare two values and returns true if there is a match and false if not. When using lists, PowerShell will return all elements in left value that match the right value.
-ne	Not Equals. Allows you to compare two values and returns true if there is not a match and false if there is. When using lists, PowerShell will return all elements in left value that don’t match

Operator	Definition
	the right value.
-ge	Greater Than Or Equal. Allows you to compare two values and returns true if the left value is greater than or equal to the right. When using lists, PowerShell will return all elements in left value that are greater than or equal to the right value.
-gt	Greater Than. Allows you to compare two values and returns true if the left value is greater than the right. When using lists, PowerShell will return all elements in left value that are greater than the right value.
-lt	Less Than. Allows you to compare two values and returns true if the left value is less than the right. When using lists, PowerShell will return all elements in left value that are less than the right value.
-le	Less Than Or Equal. Allows you to compare two values and returns true if the left value is less than or equal to the right. When using lists, PowerShell will return all elements in left value that are less than or equal to the right value.
-like	Like. Evaluates a pattern against the right value and returns true if there is a match or false if there isn´t. This operator supports wildcards, such as: • ? Any single unspecified character. • * Zero or more unspecified characters. • [a-b] Any character in a range. • [ab] The characters a or b.
-notlike	Not Like. Evaluates a pattern against the right value and returns true if there is not a match or false if there is. This operator also supports wildcards, the same as the Like operator.
-match	Match. Evaluates a regular expression against the right value and returns true if there is a match or false if there is not.
-notmatch	Not Match. Evaluates a regular expression against the right value and returns true if there is not a match or false if there is.
-contains	Contains. Returns true if a specified list has a particular value.
-notcontains	Contains. Returns true if a specified list doesn´t have a particular value.
-is	Is. Compares a value to a .NET type and return true if they match, false if they don´t.

Operator	Definition
-isnot	Is Not. Compares a value to a .NET type and return true if they don´t match, false if they do.

```
#Strings to evaluate
$a="Hello"
$b="Hi"

#EQUALS Returns False
($a -eq $b)

#NOT EQUALS Returns True
($a -ne $b)

#GREATER THAN OR EQUAL Returns True
(10 -ge 10)

#GREATER THAN Returns False
(10 -gt 10)

#LESS THAN Returns True
(3 -lt 7)

#GREATER THAN Returns True
(3 -le 7)

#LIKE Returns true
($a -like "H*")

#LIKE Returns false
($a -like "H?")

#MATCH Returns true
($a -match "(.*)")

#CONTAINS Returns true
(1,2,3,4,5 -contains 5)

#CONTAINS Returns False
(1,2,3,4,5 -contains 15)

#IS Returns true
($a -is [System.String])
```

Table 4: Logical Operators

Operator	Definition
-and	Returns true if all evaluations in a conditional statement are true. Otherwise returns false.
-or	Returns false if all evaluations in a conditional statement are false. Otherwise returns true.
-xor	Returns false if either one of the evaluations in a conditional statement is true but not if both are. Otherwise returns false.
-not	Inverts the logical evaluation in a conditional statement.

```
#AND Return False
($a -ne $b) -and (1 -eq 1) -and ($a -eq "PowerShell")
#OR Return True
($a -ne $b) -or (1 -eq 1) -or ($a -eq "PowerShell")
#XOR Returns True
($a -eq $b) -xor ($a -like "H*")
#NOT Returns False
-not ($a -ne $b)
#Combining multiple operators - Returns False
(($a -eq $b) -xor ($a -like "H*") -and (1,2,3,4 -contains 1)) -xor ("A" -ne
"B")
```

Conditional Statements

Conditional statements are similar to any other language; you have the traditional if, then, else, and the switch statement.

```
#Example
$myVar=1
if($myVar –lt 2){
    “small”
}else{
    “large”
}
```

Although the traditional declaration is available, there is another form of this statement that plays with a hash table to create what can be declared in a single line and to exactly the same as the traditional. I like to use this in simple conditions, like a value of a variable. The hash table keys need to be `$true` and `$false` because they present the two possibilities of a Boolean evaluation.

```
#@{$true=TRUE VALUE;$false=FALSE VALUE}[CONDITION]

#Example
$var1=2
$var2=2
$value = @{$true=12345;$false=67890}[$var1 -eq $var2]
#The result should be 12345
```

Switch

The switch statement is a simpler way, when compared with the if-then-else, to test multiple input cases.

```
$int = 15

switch($int)
{
      0 {“You need to sum 15”; break}
      1 {“You need to sum 14”; break}
      2 {“You need to sum 13”; break}
      3 {“You need to sum 12”; break}
      4 {“You need to sum 11”; break}
      5 {“You need to sum 10”; break}
      6 {“You need to sum 9”; break}
      7 {“You need to sum 8”; break}
      15 {“GOOD!!”; break}
      default {“You are close”;break}
}
```

Loops

In PowerShell, you have several ways to loop (iterate) an object. However, the possibilities you have depend on your logic, code organization goals, and your object type. The simplest of them is the `while` loop. Using a while in PowerShell is the same as in any other C-based language.

```
#EXAMPLE

$array = 1,2,3,4,5,6

$var=0

while($var -lt 6){

      write $array[$var]

      $var++
```

```
}
```

Another looping structure you can use is the “for loop”, which again is similar to any C based language.

```
#EXAMPLE

$array = 1,2,3,4,5,6

for($i=0;$i -lt $array.Count;$i++){
      write $array[$i]
}
```

The last is one of the most used in PowerShell, the for each loop structure, because of its appliance to pipeline objects. However, there are two different kinds of for each structures, one to iterate in collections of objects that implement IEnumerable and another for input object collections, often used to iterate pipeline objects.

As you will see in the following code block, by using the PowerShell foreach input object, code becomes simpler and cleaner. The results are exactly the same.

```
#BASE
$array = 1,2,3,4,5,6

#Traditional way - classic foreach
foreach($var in $array){
      write $var
}


<# Using the most common way to
      iterate objects in PowerShell
      - input object foreach
#>
$array | %{
      write $_
}
```

Tip: The foreach object has commonly used aliases, the character % (Percent) and foreach. The first in most script editors assume a green color and the second a blue one.

Managing the Flow

PowerShell lets you manage delays, pauses, and breaks in your scipts. To do that, you have a set of commands to help you. The `Read-Host` command pauses your scipt until the user presses the Enter key. The `Start-Sleep` command pauses your scipt for a given amount of time.

```
#Sleeps for 15 seconds.
Start-Sleep -Seconds 5

#Pause until user presses the Enter key.
Read-Host
```

Schedule Script Execution

Scheduling a script execution is a basic operation to automate a specific task in PowerShell. It could be to store a log, either for an integration task or even creating a backup for an SQL database table. There are plenty of situations in which you might think about letting PowerShell automate things for you.

Although you might be thinking about coding a Windows Communication Foundation (WCF) or something else, there is an easy strategy to schedule a script execution using Windows Task Scheduler. This allows you to separate the script development from its execution management. To use it, start by opening it in Windows (**Start** > **All Programs** > **Accessories** > **System Tools** and select **Task Scheduler**). Once it opens, select **Create Basic Task**.

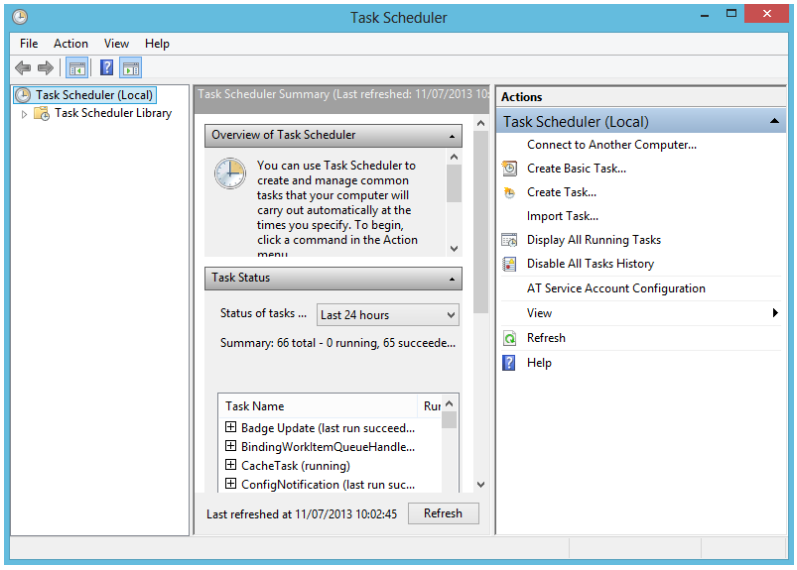


Figure 21: Open Task Scheduler

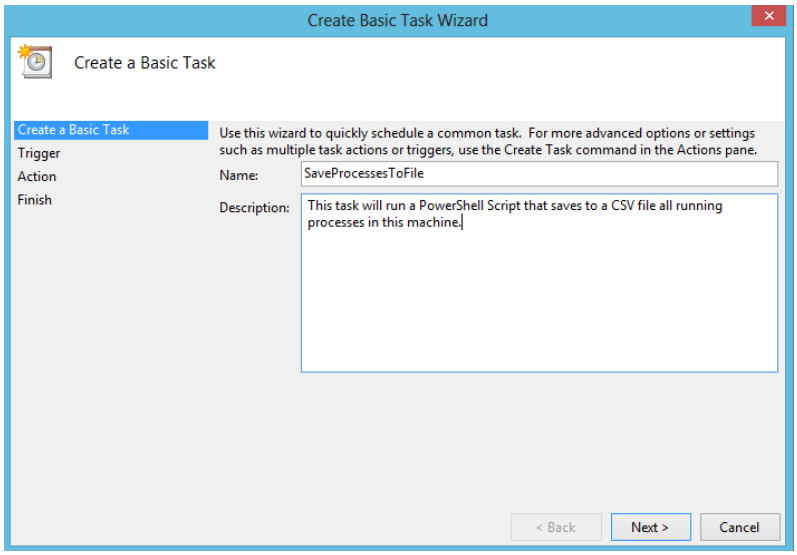


Figure 22: Create Basic Task

Now select when you want to run this task. This is just the first schedule configuration, as you need to specify later more details about this execution. For example, if you choose Daily, you can define the hours at which this script will run.

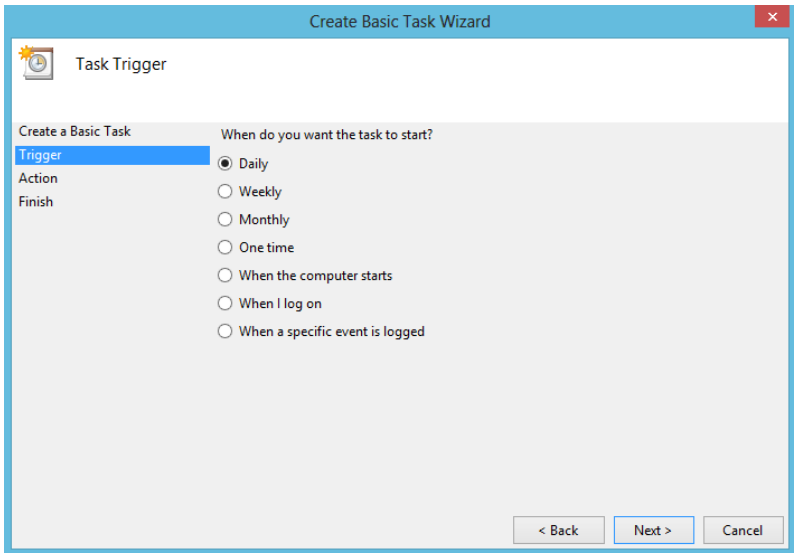


Figure 23: Scheduling Selection

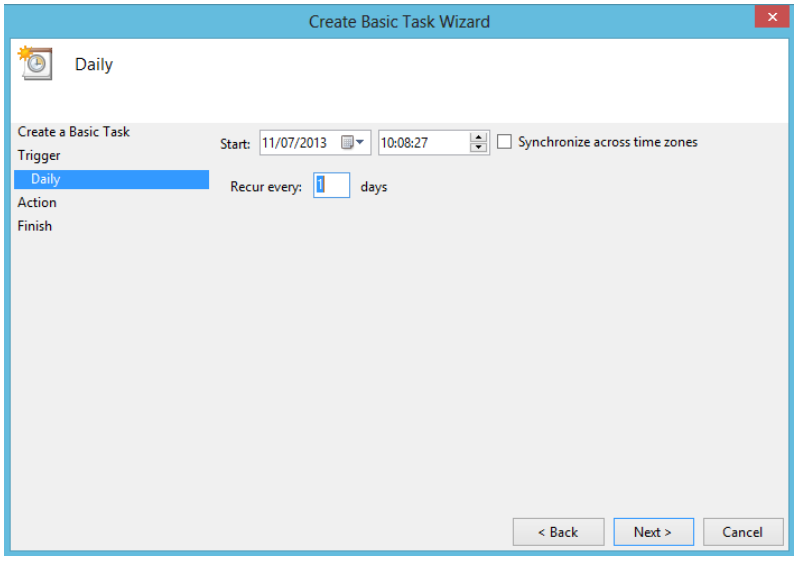


Figure 24: Second level Scheduling

After you define the execution schedule you choose the action, which in this case you must select **Start a Program**. On the program/script selection, type **PowerShell.exe** in the program textbox and as an argument type **–file** and after the path to the script you want to run. After that you are done. The script will execute at the selected time with a plus, you can change the script whenever you want and the changes are automatically available.

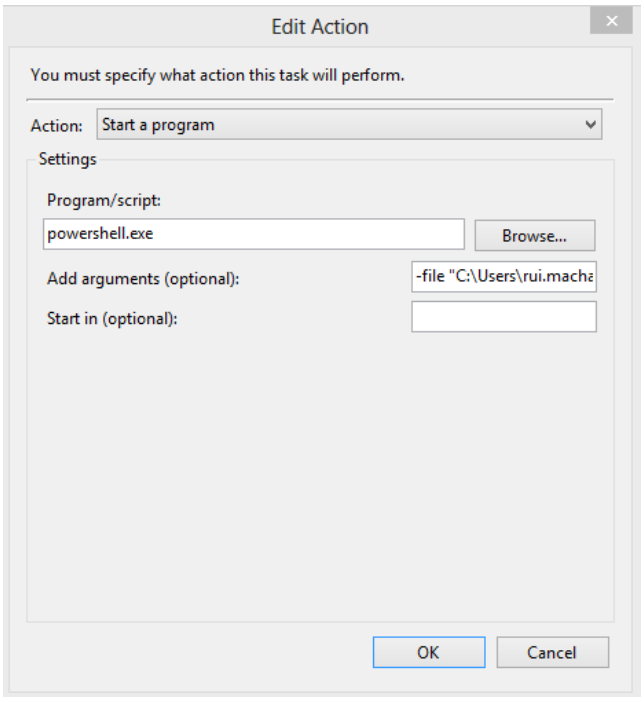


Figure 25: Finish Schedule Task

Extensibility and Code Reuse

Create instances of objects

Creating instances of objects in PowerShell is easy thanks to the new `New-Object` cmdlet. You can invoke `New-Object` indicating which object you want to instantiate, whether it's a .NET or COM object.

```
#example using New-Object and ArgumentList to pass parameter to the object
constructor
$eventLog = New-Object System.Diagnostics.EventLog -ArgumentList "Application"

#Object is available.
$eventLog.Entries | Out-GridView
```

Functions and Parameters

Let's start by taking a look at functions. Functions are named code blocks, preferably with simple and single tasks, built to reuse code within your scripts. Parameters, on the other hand, are a simple mechanism of communication to send external values to a function. In the following code sample, you can find several application scenarios for functions and parameters.

```
#Using a simple Function
function GetAllProcess{
      Get-Process
}

#Using a simple Function two parameters.
function MyFunction($param1,$param2){
      Write "This is param 1 : $param1"
      write "This is param 2 : $param2"
}

#You can also use the param block to define your parameters.
function MyFunction{
      param(
            $param1
            ,$param2
      )
      #Even declare where you process your logic.
      process{
            Write "This is param 1 : $param1"
            write "This is param 2 : $param2"
      }
}

#Calling GetAllProcess
```

```
GetAllProcess

#Calling MyFunction
MyFunction -param1 65 -param2 3
```

Tip: As you can see in the previous code block, you can declare parameters inside the param block as well as declare your logic inside of a process block. This will make your code more readable and better organized.

When calling functions, there are two ways to pass parameters into them, either by including them in the function signature or simply displaying the value in front of the function call.

```
#Included in method signature.
[string]::Equals("Rui","Machado")

#Using PowerShell front function parameters.
MyFunction -param1 65 -param2 3
```

Tip: You should be careful with this parameter passing. The use of parentheses in function calls or cmdlets is interpreted as a sub-expression of PowerShell and not as a parameter. In this situation, the parameters must be passed by specifying the parameter in question. We can only use parameter passing in methods within the method signature (between parentheses). In the following code sample, you can see the differences between these situations.

```
function SumTwoValues($a,$b)
{
  return $a + $b
}

<#CASE 1
Calling a function with its parameters inside its signature
doesn´t give you the expected result (evaluating the parameters as
a sub expression), which in this case is an array. In PowerShell you can
declare an array as 1,2,3,4,5,...,n, so it prints the values of the array
#>

"CASE 1"
SumTwoValues(1,4)

<#CASE 2
Correct calling: Using this method you will get the expected result
#>

"CASE 2"
SumTwoValues -a 1 -b 5
```

Figure 26 shows the results produced when calling the SumTwoValues method using both syntactical options. CASE 1 displays the array (1,4) and CASE 2 displays the sum of 1 and 5.

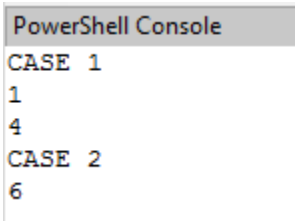


Figure 26: Passing Parameters in Functions

Parameters can be used not only in functions but in scripts as well. PowerShell allows you to use script parameters, making the entire script reusable just by calling it as a function, which is very useful when you want to create a data flow based on scripts instead of functions. To add parameters to your scripts, you need to use the block `param` at the beginning of your script, which will only work if it is the first line of code in your script.

```
Param(
      $param1,
      $param2
)

#Start your script definition.
```

As you can see, parameters are defined inside that param block, but there is much more to tell about these parameters and their special attributes. In this param block, as in any function, you can add the following attributes.

Table 4: Parameter Attributes

Attribute	Definition
Mandatory (Boolean)	Specifies if the parameter is mandatory; True for yes, False for no.
ParameterSetName (String)	Specifies the parameter set that the command parameter belongs to.
Position (Integer)	Specifies the position in which the parameter is in that command signature.
ValueFromPipeline (Boolean)	Specifies if the parameter comes from a pipeline object. Default value is False.
ValueFromPipelineByPropertyName (Boolean)	Specifies if the parameter cames from a property of a pipeline object. Default value is False.
ValueFromRemainingArguments (Boolean)	Specifies that the parameter accepts any value from remaining command parameters.

Attribute	Definition
HelpMessage	Set a description for the parameter.
HelpMessageBaseName	Specifies an external reference for the Help message description.
HelpMessageResourceId	Specifies the resource identifier for a Help message.

To use any of these parameters, you need to declare them between square brackets right before or on top of the parameter declaration. The following code block shows an example of this attributes usage.

```
function TestParameters{
      param(
            #You need to write the attributes inside Parameter()

            #That parameter is Mandatory
            [Parameter(Mandatory=$true)] $name,

            #That parameter is Mandatory and Add a Help Message
            [Parameter(Mandatory=$true,HelpMessage="You must enter an
Age!")]
            $age,

            #This parameter comes from a pipeline object.
            [Parameter(ValueFromPipeline =$true)] $address
      )
}
```

As you can see in Figure 27, by using a help message, the user can see more information about a specific parameter when trying to invoke the function.

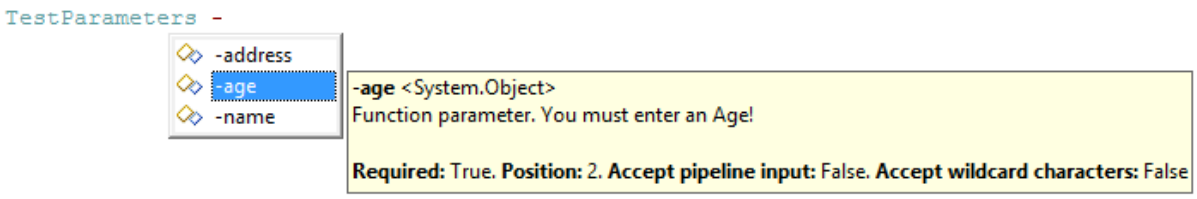


Figure 27: Help Message

If you don´t provide a value from a mandatory parameter, PowerShell will raise an input box telling you that mandatory parameters are missing and asking you to provide them.

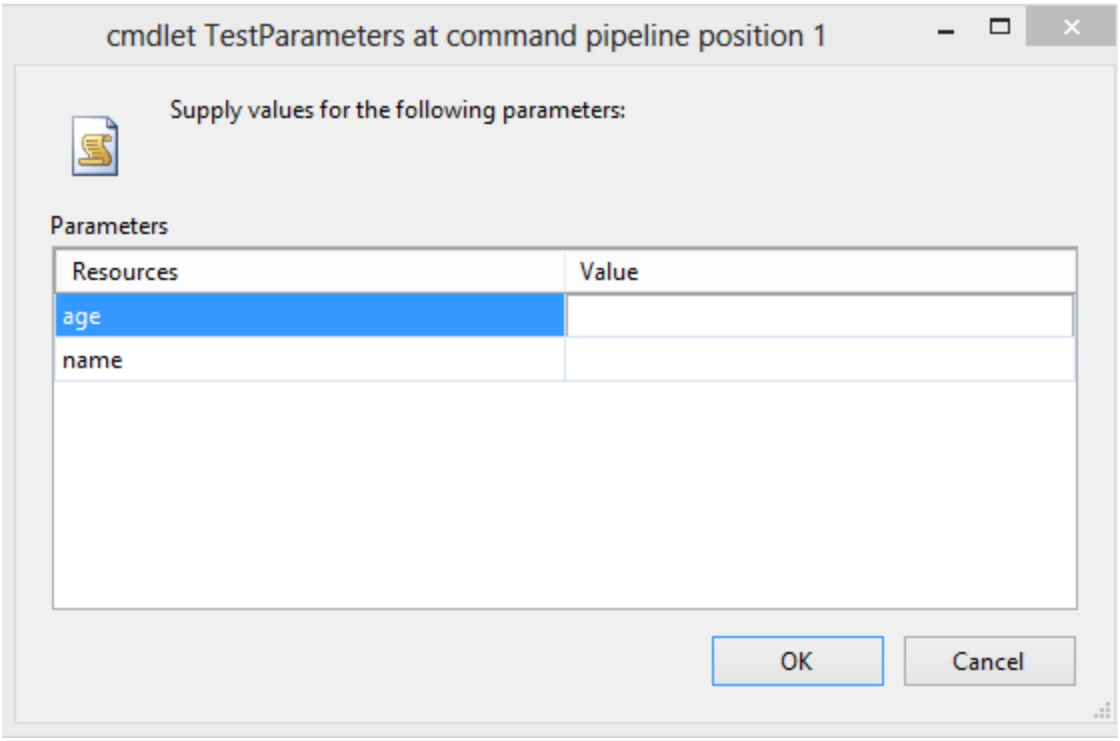


Figure 28: Missing Parameters

Another nice feature about parameters is that you can add validators to evaluate if your parameters match a validation rule set by you.

Note: There are several validators for PowerShell and it's important that you know about them, as they will save you time with validations on values for particular parameters. The following table defines every possible validator.

Validator	Definition	Definition
ValidateCount	Validates the minimum and maximum parameters allowed for a command.	`[ValidateCount($min,$max)]`
ValidateLength	Validates the minimum and maximum number of characters for a parameter	`[ValidateLength($min,$max)]`
ValidatePattern	Validates a string parameter with a regular expression.	`[ValidatePattern($regexString)]`
ValidateSet	Validates a parameter	`[ValidateSet($arrayValidValues)]`

Validator	Definition	Definition
	according to a set of possible values.	
ValidateRange	Validates a parameter according to a range of possible values.	`[ValidateRange($min, $max)]`

The following example uses the `ValidateSet` attribute to enforce a constraint on the $type parameter, which limits the values that can be passed to that parameter to three distinct values. The `ChangeUserType` function then uses that parameter to change the user type.

```
function ChangeUserType{
      param(
            $userID,
            [ValidateSet("Admin","User","SuperAdmin")]
            $type
      )
      ChangeType -User $userID -Type $type
}
```

If you try to call that function with a type that is not specified in the validation set, you will get an error indicating that a particular value is not defined for that parameter.

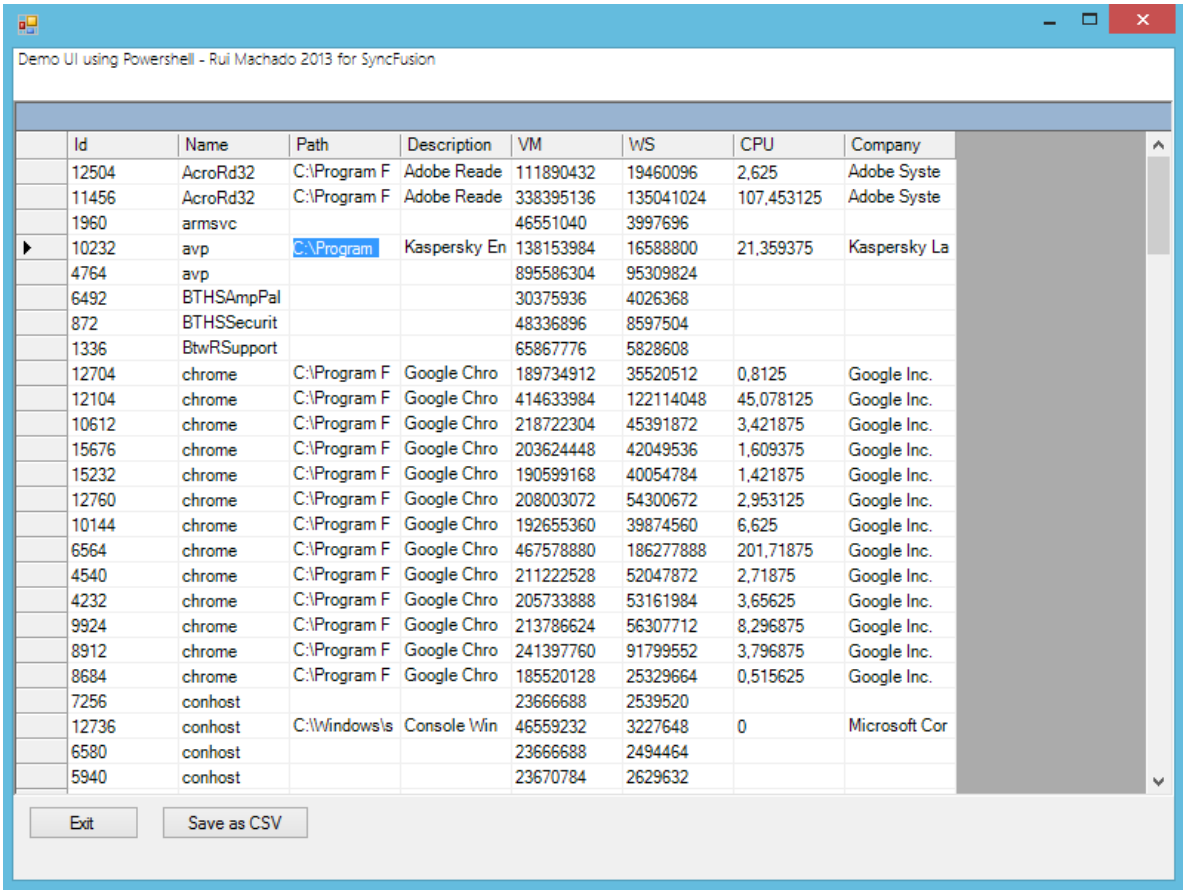
```
ChangeUserType -userID 1 -type "Regular"
```

```
ChangeUserType : Cannot validate argument on parameter 'type'. The argument "Regular" does
not belong to the set "Admin,User,SuperAdmin" specified by the ValidateSet attribute.
Supply an
argument that is in the set and then try the command again.
At C:\Users\rui.machado\Dropbox\PowerSHell Book\Scripts\params.ps1:16 char:32
+ ChangeUserType -userID 1 -type "Regular"
+                                ~~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [ChangeUserType],
ParameterBindingValidationException
    + FullyQualifiedErrorId : ParameterArgumentValidationError,ChangeUserType
```

Figure 29: Error with Attribute outside ValidationSet

Create a Windows Form

Creating Windows Forms in PowerShell is as simple as in C#. By having the ability to interact with .NET classes, we just need to reference the class in which our Windows Form Item is defined and then instantiate it using the method shown in the previous topic, Create instances of objects. To show you how simple it is, I am going to build an example in which I create a form with a header label and a data grid to list items from our `Get-Process` command. Next I will add a button that will invoke a save file dialog so that we can export our data grid data source as a CSV file. The expected result is shown in Figure 30.



Id	Name	Path	Description	VM	WS	CPU	Company
12504	AcroRd32	C:\Program F	Adobe Reade	111890432	19460096	2,625	Adobe Syste
11456	AcroRd32	C:\Program F	Adobe Reade	338395136	135041024	107,453125	Adobe Syste
1960	armsvc			46551040	3997696		
10232	avp	C:\Program	Kaspersky En	138153984	16588800	21,359375	Kaspersky La
4764	avp			895586304	95309824		
6492	BTHSAmpPal			30375936	4026368		
872	BTHSSecurit			48336896	8597504		
1336	BtwRSupport			65867776	5828608		
12704	chrome	C:\Program F	Google Chro	189734912	35520512	0,8125	Google Inc.
12104	chrome	C:\Program F	Google Chro	414633984	122114048	45,078125	Google Inc.
10612	chrome	C:\Program F	Google Chro	218722304	45391872	3,421875	Google Inc.
15676	chrome	C:\Program F	Google Chro	203624448	42049536	1,609375	Google Inc.
15232	chrome	C:\Program F	Google Chro	190599168	40054784	1,421875	Google Inc.
12760	chrome	C:\Program F	Google Chro	208003072	54300672	2,953125	Google Inc.
10144	chrome	C:\Program F	Google Chro	192655360	39874560	6,625	Google Inc.
6564	chrome	C:\Program F	Google Chro	467578880	186277888	201,71875	Google Inc.
4540	chrome	C:\Program F	Google Chro	211222528	52047872	2,71875	Google Inc.
4232	chrome	C:\Program F	Google Chro	205733888	53161984	3,65625	Google Inc.
9924	chrome	C:\Program F	Google Chro	213786624	56307712	8,296875	Google Inc.
8912	chrome	C:\Program F	Google Chro	241397760	91799552	3,796875	Google Inc.
8684	chrome	C:\Program F	Google Chro	185520128	25329664	0,515625	Google Inc.
7256	conhost			23666688	2539520		
12736	conhost	C:\Windows\s	Console Win	46559232	3227648	0	Microsoft Cor
6580	conhost			23666688	2494464		
5940	conhost			23670784	2629632		

Figure 30: Windows Form Get-Process

```
#Create a new .NET array list.
$processList = New-Object System.Collections.ArrayList

#Get all processes.
$allProcesses = Get-Process | Select
Id,Name,Path,Description,VM,WS,CPU,Company | sort -Property Name

#Add all processes to an array list which is easier to manipulate.
$processList.AddRange($allProcesses)

#Instantiate a new Windows Form
$form = New-Object Windows.Forms.Form

#Sets the Windows Form size and start position.
$form.Size=New-Object Drawing.Size @(800,600)
$form.StartPosition = [System.Windows.Forms.FormStartPosition]::CenterScreen

 #This will create panels to display our items.
$panelLabel = New-Object Windows.Forms.Panel
$panelMain = New-Object Windows.Forms.Panel
$panelButton = New-Object Windows.Forms.Panel
```

```
#Creates the save file dialog so that we can export it as CSV.
$saveDialog = new-object System.Windows.Forms.SaveFileDialog
$saveDialog.DefaultExt = ".csv"
$saveDialog.AddExtension = $true

#Create the save to CSV button.
$buttonSave = New-Object Windows.Forms.Button
$buttonSave.Text = "Save as CSV"
$buttonSave.Left = 100
$buttonSave.Width =100

#Add the OnClick save button event.
$buttonSave.add_Click(
{
      $resultSave=$saveDialog.ShowDialog()
      #If the user clicks ok to save.
      if($resultSave -eq "OK"){
            #Save as CSV
            $allProcesses | Export-Csv -Path $saveDialog.FileName
            MessageBox("Guardado com sucesso")
      }
})

#Create the exit application button.
$button = New-Object Windows.Forms.Button
$button.Text = "Exit"
$button.Left = 10

#Add the OnClick exit button event.
$button.add_Click(
{
      $form.Close()
})

#Create datagrid
$dataGrid=New-Object Windows.Forms.DataGrid
$dataGrid.Dock = "Fill"
$dataGrid.DataSource = $processList

#Create a new label to show on the header.
$label = New-Object System.Windows.Forms.Label
$label.Text= "Demo UI using Powershell - Rui Machado 2013 for SyncFusion"
$label.Font = "Segoe UI Light"
$label.Width= 300

#Add the header label to its panel.
$panelLabel.Controls.Add($label)
$panelLabel.Height =35
$panelLabel.Dock = "Top"
```

```
$panelLabel.BackColor = "White"

#Add datagrid to its panel.
$panelMain.Controls.Add($dataGrid)
$panelMain.Height =470
$panelMain.Dock = "Top"

#Adds buttons to its panel.
$panelButton.Controls.Add($button)
$panelButton.Controls.Add($buttonSave)
$panelButton.Height=50
$panelButton.Dock = "Bottom"

#Add all panels to the form.
$form.Controls.Add($panelMain)
$form.Controls.Add($panelButton)
$form.Controls.Add($panelLabel)

$form.Refresh()

#Show the form.
$result = $form.ShowDialog() | Out-Null

if($result -eq "Cancel")
{
      MessageBox("Program is closing...")
      $form.Close()
}

#OPTIONAL: Function to create new MessageBoxes
function MessageBox([string]$msgToShow)
{
      [System.Windows.Forms.MessageBox]::Show($msgToShow)
}
```

Clicking on the **Save CSV** button will save the file to a destination in your file system.

Chapter 2 File System

This chapter is specifically for system administrators who dedicate several hours in their daily routine to working with files and directories. PowerShell won´t free you from doing it, but it might help you automate your manual tasks by writing reusable functions.

Current Location

An important topic when dealing with files and directories is setting your paths. Current location is one of those important paths for you to create relative paths between files and even scripts. To get your current location, you use the `Get-Location` command, which retrieves an object that represents the current directory, much like the pwd (print working directory) command. In this object you have the current path and drive.

```
#Returns full path for the current location.
Get-Location | %{$_.Path}

#Return drive info.
Get-Location | %{$_.Drive}
```

```
PS C:\Windows\system32> Get-Location | %{$_.Path}
C:\Windows\system32
PS C:\Windows\system32> Get-Location | %{$_.Drive}

Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
C                 144,50         77,72 FileSystem    C:\
```

Figure 31: Get-Location Execution

Get Files from a Directory

To retrieve information about all files in a directory, you can use the `Get-ChildItem` command, or if you just want to list information about a single item, use the `Get-Item` command. This command has several important parameters; three of them are highlighted in Table 5.

Table 5: Get-Item parameters

Parameter	Definition
-Name	Gets only the names of the items in the locations.
-Recurse	Gets the items in the specified locations and in all

Parameter	Definition
	child items of the locations.
-Path	Specifies a path to one or more locations.

```
#Simple Get-ChildItem
Get-ChildItem -Path (Get-Location).Path

#Get-ChildItem using -Recurse parameter.
Get-ChildItem -Path (Get-Location).Path -Recurse

#Get-ChildItem using -Recurse parameter and a filter for file name (-Name)
Get-ChildItem -Path (Get-Location).Path -Name "*Rui*"

#Get-ChildItem filter by extension.
Get-ChildItem -Path (Get-Location).Path | ?{$_.Extension -like "*txt"}
```

Get the Content of a File

```
#Get the content of a single file without restrictions.
Get-Content -Path "c:\temp\File.txt"
```

Getting the content of a file is an easy task thanks to the `Get-Content` command, which will read the content one line at a time and return a collection of objects, each of which represents a line of its content. You can either invoke this command alone or combine it with the `Get-ChildItem` command to dynamically read the content of multiple files, according to its attributes such as last change date.

The previous code block shows you a simple example on how to retrieve the content of a single file. This will result in an object array with all lines red, which you can save into a variable and manipulate later. Figure 32 reproduces the result of this calling. To test this code block, create a file named **File.txt** in **C:\temp**.

```
PS C:\Users\rui.machado> #Get the content of a single file without restrictions
PS C:\Users\rui.machado> Get-Content -Path "c:\temp\File.txt"
Hello PowerShell Fans
This is a Get-Content example
Made for SyncFusion Succinctly Series
By Rui Machado
```

Figure 32: Calling Get-Content

Your requirements might involve more elaborate callings. In the following code block there are several examples on how to use this command.

```
#Get the first three lines of your file using the -TotalCount parameter.
Get-Content -Path "c:\temp\File.txt" -TotalCount 3
```

```
#Get the last line of your file using the -Tail parameter (Available since
Powershell v3.0)
Get-Content -Path "c:\temp\File.txt" -Tail 1

<#
      In this example combining this command with Get-ChildItem
      will allow you to retrieve the content of all files in c:\temp
directory
#>
#Get the files
Get-ChildItem -Path "c:\temp" -Filter "*.txt" | %{
      #Get Content
      Get-Content -Path $_.FullName
}
```

Manipulate the Content of a File

Once you have the content of a file, it is likely that you will want to change its content, either by cleaning it, replacing strings, or even adding text to it. The `Set-Content` command provides this ability by allowing you to write or replace the content of a specified item, such as a file. If you use this command alone and invoke it with only a path and value, then it will replace the entire content with the new one specified in the value parameter. To replace text in a file, you need to combine it with the Get-Content command.

In the following example, we will get the content of a file and replace every instance of the string "_(TOCHANGE)" with the string "PowerShell". The original text inside the file is the following:

"_(TOCHANGE) Hello _(TOCHANGE) Fans. This is a Get-Content example made for Syncfusion Succinctly Series by Rui Machado"

Now we will run the following script to change the file content to the correct sentence. Note that what will change the content of the file is the string replace made after reading the content of the file. The Set-Content command will just take the path of the file and the new value for its content passed through the pipeline.

```
#File path.
$path = "c:\temp\File.txt"

#Get the file content.
Get-Content -Path $path | %{

      #Replace the token in memory, this won't change the file content.
      $new = $_.Replace("_(TOCHANGE)","PowerShell")
}
#Set the file content.
Set-Content -Path $path -Value $new
```

This will result in a content reset for our file as you can see in the following figure.

```
PS C:\Users\rui.machado> Get-Content -Path $path
_(TOCHANGE) Hello _(TOCHANGE) Fans. This is a Get-Content example made for SyncFusion Succinctly Series by Rui Machado
PS C:\Users\rui.machado> #file path
PS C:\Users\rui.machado> $path = "c:\temp\File.txt"
PS C:\Users\rui.machado>
PS C:\Users\rui.machado> #Get the file content
PS C:\Users\rui.machado> Get-Content -Path $path | %{
>>
>> #Replace the token in memory, this wont change the file content
>> $new = $_.Replace("_(TOCHANGE)","PowerShell")
>> }
>> #Set the file content
>> Set-Content -Path $path -Value $new
>>
PS C:\Users\rui.machado> Get-Content -Path $path
PowerShell Hello PowerShell Fans. This is a Get-Content example made for SyncFusion Succinctly Series by Rui Machado
PS C:\Users\rui.machado>
```

Figure 33: Reset file content

Create Temporary Files

Using temporary files is a useful strategy for manipulating files across several script invocations. For instance, you can create a temporary XML file and manipulate it using several function and method calls and then save it only at the end of your script execution.

To create a temporary file, you can use the .NET namespace that handles paths, `[System.IO.Path]` which among other things includes a method called `GetTempFileName()`for creating temporary files.

```
#This gives you a temp url file to work with.
$url = [System.IO.Path]::GetTempFileName()

#After getting the temp url, you can work with the file.
"Rui" | Out-File $url

#You can retrieve the content of the temp file.
Get-Content $url
```

Manage Directories

Create New Directories

Managing directories includes tasks such as creating new directories, deleting directories, changing permissions, and renaming directories. PowerShell allows you to use the common MS-DOS commands to manage directories, but it adds new commands so that managing directories becomes easier, faster, and more efficient.

To create a new directory in the old Windows shell you use mkdir. PowerShell redefines the mkdir command by offering you the `New-Item` command. The `New-Item` command is not only used to create directories, but can also be used to create files, registry keys, and entries. `New-Item` takes a –type parameter that determines the type of item to create.

To create a new directory, you just need to invoke the command `New-Item`, providing as parameters its destination path and the type directory parameters.

```
New-Item  -Path "c:\temp\newFolder" -ItemType Directory
```

```
PS C:\Users\rui.machado> New-Item  -Path "c:\temp\newFolder" -ItemType Directory


    Directory: C:\temp


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d-----        09/07/2013     20:51            newFolder
```

Figure 34: Create a new directory

Change Directory Permissions

Although the directory is already created, you might want to change its permissions. To do so, you must be aware of the access control list (ACL) concept, which is a list of access control entries (ACE) that identifies a trustee and specifies the access rights allowed, denied, or audited for that trustee. In another words, it specifies the permissions that users and user groups have to access a specific resource.

PowerShell allows you to get and set the ACL for a file or directory, and to do so you can use the `Get-Acl` command to retrieve objects that represent the security descriptor of a file or resource and the `Set-Acl` command to change the security descriptor of a specified item. The following code block shows how you can change the "newFolder" created previously to deny access to the users group.

```
$pathToFolder = "c:\temp\newFolder"

#Get the ACL of the folder.
$acl = Get-Acl -Path $pathToFolder

#set the new permission settings.
$perSettings = "BUILTIN\Users","FullControl","Deny"

#create the access rule.
$newRule = New-Object System.Security.AccessControl.FileSystemAccessRule
$perSettings

#change the acl access rule.
$acl.SetAccessRule($newRule)

#Set the new rules in the folder ACL.
$acl | Set-Acl $pathToFolder
```

This will add a new entry in the permissions for this folder as you shown in Figure 35.

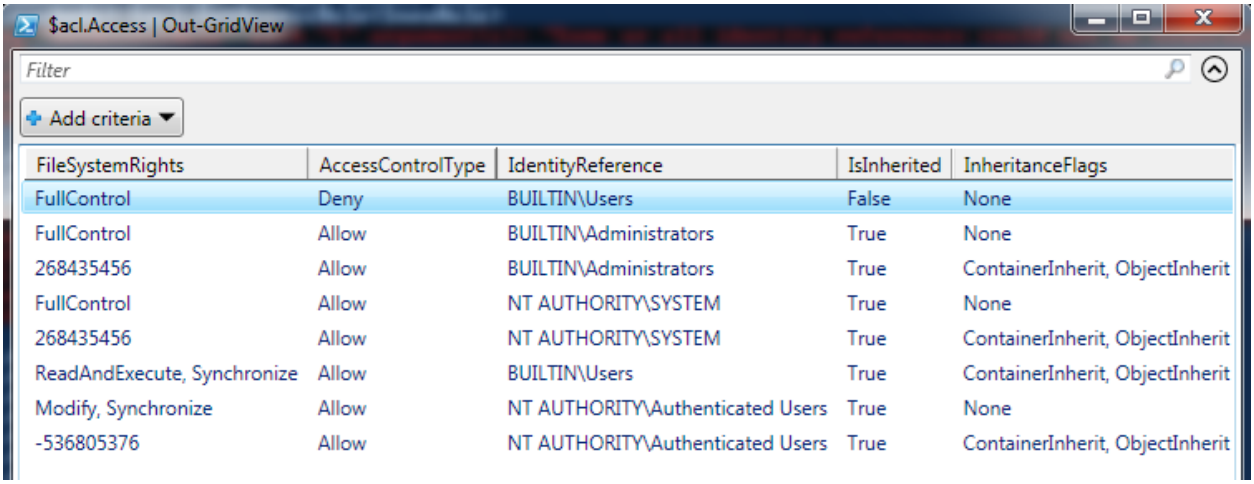


Figure 35: Change directory permissions

To get the output displayed in the previous figure, run the following code:

```
$acl = Get-Acl -Path $pathToFolder
$acl.Access | Out-GridView
```

Remove Directories

To remove a directory, you can use the command `Remove-Item` which is similar in behavior and calling structure to the `New-Item` command. In this case you just need to provide the item path without providing the item type.

```
Remove-Item  -Path "c:\temp\newFolder"
```

Combining the `Get-ChildItem` with `Remove-Item` command will allow you to create mechanisms of batch deletion of files inside directories according to a filter or file attribute.

The last manipulation task I want to talk about is renaming your files and directories. PowerShell also offers you a command to rename your files, which can be combined with the `Get-ChildItem` command, if you are working with sets of files in order, to rename several at once according to a given filter or file attribute.

Rename Directories

The `Rename-Item` command changes the name of a specified item. This command does not affect the content of the item being renamed. To use it, you just need to provide as a parameter the path to the destination directory or file and the new name to set.

```
#directory path
$path = "c:\temp\"
```

```powershell
#This example will invoke the rename command to a set of files.
Get-ChildItem -Path $path -Name "File.txt" | %{

      #Rename the file.
      Rename-Item -Path  "$path\$_" -NewName "FileRenamed.txt"
}

#Simple use of rename item command.
Rename-Item -Path "$path\FileRenamed.txt" -NewName "File.txt"
```

Move a File or a Directory

To move a file or a directory, you can use the `Move-Item` command, which moves an item, including its properties, contents, and child items, from one location to another location. An important note to retain is that when you move an item, it is added to the new location and deleted from its original location.

```powershell
#original file location. Where we want to move from.
$original = "c:\temp\File.txt"

#file target location. Where we want to move to.
$destination = "c:\temp\newFolder\File.txt"

#Invoke the move item command.
Move-Item -Path $original -Destination $destination

<#Path and destination are positional parameters 1 and 2 so you can ignore
parameter referencing.
     Move-Item $original $destination
#>

#Move the entire directory.
Move-Item "c:\temp\newFolder" "c:\"
```

Managing Paths

Join Parts into a Single Path

PowerShell comes out of the box with great mechanisms to manage paths in your scripts, using operations like creating full paths by joining its relative parts, testing if a path exists, completing a relative path, or even splitting a path into its multiple parts.

For the task of joining multiple path parts, you can use the `Join-Path` command, which combines a path and child path into a single path. The provider supplies the path delimiters.

```
#This will return c:\temp\File.txt
Join-Path -Path "c:\temp" -ChildPath "File.txt"

#Because path and childpath are positional parameters (Path is 1 and
ChildPath is 2) ignore the parameter reference.
Join-Path "c:\temp" "File.txt"

#Join multiple parts.
Join-Path (Join-Path "c:" "temp") "File.txt"
```

Split Paths into multiple parts

But what about the opposite task, splitting a path into multiple parts? To do that, you need to use the `Split-Path` command. This will return the specified part of a path that you defined in your command call, either the leaf (the last part of the part) or the head (the default behavior).

```
$path = "c:\temp\File.txt"

#Return the head of the path: c:\
Split-Path -Path $path

#Return the leaf of the path: File.txt
Split-Path -Path $path -leaf

#Split multiple times: temp
Split-Path (Split-Path -Path $path) -leaf
```

To enrich your split path command, you can use some .NET methods to get a file name or extension, a behavior that Split-Path won't support.

```
$path = "c:\temp\File.txt"

#Return the file name: File.txt
[System.IO.Path]::GetFileName($path)

#Return the file extension: .txt
[System.IO.Path]::GetExtension($path)

#Return the file name without extension: File
[System.IO.Path]::GetFileNameWithoutExtension($path)

#Return the full file name: c:\temp\File.txt
[System.IO.Path]::GetFullPath($path)

#Return the directory name: c:\temp
[System.IO.Path]::GetDirectoryName($path)
```

Test if Path Exists

While accessing a particular file, you may want to change or get its content. It is a good idea to check and see if the file or directory exists. To accomplish this, PowerShell has the `Test-Path` command, which determines whether or not all elements of the path exist. It returns true ($true) if all elements exist and false ($false) if any are missing. It can also tell whether the path syntax is valid and whether the path leads to a container or a terminal (leaf) element.

This command is often used in combination with other commands as you see in the following code block.

```
$path = "c:\temp\File.txt"

#Simple path test, returns: True
Test-Path -Path $path

#You can evaluate the path forcing it to check if it is a file: True
Test-Path -Path $path -PathType Leaf

#You can evaluate the path forcing it to check if it is a directory: False
Test-Path -Path $path -PathType Container

#combine it with the Get-Content commands.
$found=(Test-Path -Path $path -PathType Leaf)
#if it is found, retrieve the content of the file.
if($found){
      Get-Content $path
}
```

Resolve Paths

The last task I want to mention in this managing path section is the ability to resolve paths dynamically in PowerShell. Resolving paths means interpreting wildcard characters in a path and displaying the path contents. To do this, you need to use the `Resolve-Path` command, which interprets the wildcard characters given and displays the items and containers at the location specified by the path, such as the files and folders or registry keys and sub keys.

```
#Target Directory
$path = "c:\temp\*"

#Get all files inside that directory.
Resolve-Path -Path $path

#Once again path is a positional parameter in position 1, so ignore
parameter reference.
Resolve-Path $path
```

By calling the previous script, PowerShell will list, according to the wildcard defined, all files inside the given path.

```
PS C:\Users\rui.machado> #Target Directory
PS C:\Users\rui.machado> $path = "c:\temp\*"
PS C:\Users\rui.machado>
PS C:\Users\rui.machado> #Get all files inside that directory
PS C:\Users\rui.machado> Resolve-Path -Path $path

Path
----
C:\temp\newFolder
C:\temp\diskLog.txt
C:\temp\File - Copy.txt
C:\temp\File.txt
C:\temp\File2.txt
C:\temp\ListaUsers.csv
C:\temp\test.xlsx
C:\temp\testes.xml
C:\temp\tests.xml
```

Figure 36: Result of Resolve-Path calling

Chapter 3 Processes

List All Processes

To list every process, as shown in previous examples, use the `Get-Process` command. Just by invoking the command in the shell, it will internally call a `foreach` loop to show you every active process in your computer with some properties and show it as a formatted table.

```
Get-Process
```

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    406      43   109220     131808   323   106,39  11456 AcroRd32
    385      24     9540      18904   107     2,28  12504 AcroRd32
     75       7     1124       3856    44           1960 armsvc
    394      18    10668      14864    78   468,02   1512 audiodg
   2980     236   386608      85408   823           4764 avp
   1211      48    10872      15740   128    15,72  10232 avp
     55      11     1120       3908    29           6492 BTHSAmpPalService
    228      26     3712       8380    46            872 BTHSSecurityMgr
    105      10     1696       5672    63           1336 BtwRSupportService
    156      26    27212      33208   184     0,59   2688 chrome
   2475      84   179700     196576   467   132,48   6564 chrome
    156      28    29664      36288   190     0,72   7656 chrome
    159      31    30492      37676   182     9,94   7712 chrome
```

Figure 37: Get Process Result

There are some parameters and options you might want to use to list all active processes according to requirements you may have. In the following example, you should replace the `MyMachineName` string value with a valid machine name. You can check your machine name by running the `hostname` command in your shell or command-line tool.

```
#Execute Get-Process on a remote computer.
Get-Process -ComputerName "MyMachineName"

#Get specific attributes.
Get-Process | select Id,Name

#Condition the process you want to list (Chrome processes only).
Get-Process | ?{$_.Name -like "*chrome"}
```

Another thing you can do after retrieving all running processes is to process them to extract information. An example of that is evaluating CPU usage for all Google Chrome processes on your machine.

```
#How much CPU do all chrome processes consume?
$sumCpuUsage=0
write "All Chrome Processes:"
Get-Process -ComputerName "MyMachineName”  |`
      ?{$_.Name -like "*chrome"} | %{
                  $id=$_.Id
                  $cpu = $_.CPU
                  "Process:$id  CPU: $cpu"
                  $sumCpuUsage+=$_.CPU
            }
write "Total CPU: $sumCpuUsage"
```

Get a Process by ID

After listing all your processes, you might identify a process that you wish to analyze in particular; to do so, you must get its ID and then call it using the attribute –Id of the `Get-Process` command.

```
Get-Process -Id 11372
```

Stop a Process

To stop a process, you need to use the command `Stop-Process` followed by the attribute ID, which is mandatory in this situation. To optimize your command, you might get that process by ID and then pipe it to the stop command. In the following example, you find all instances of Internet Explorer running in your machine and then stop them.

```
#Using the get by Id concept
Get-Process -ProcessName “iexplore”  | Stop-Process
```

Start a Process

Launching a process in PowerShell is easy thanks to the `Start-Process` command. This command not only starts the process but also returns the process object itself, which can then be examined or manipulated. This command starts one or more processes on the local computer, either if you set as a parameter the path to the executable file, or if you set as a path, one to a file. In this case, PowerShell will launch the default program to open the file’s extension.

```powershell
<# EXAMPLE 1: Start a program #>

      #Path from file.
      $path = "C:\temp\Paint.NET.lnk"

      #Start new process.
      $process = Start-Process $path

      #Get process id started.
      $process.Id


<# EXAMPLE 2: Start a file
with the default program to open it #>

      #Path from file.
      $path = "C:\temp\ListaUsers.csv"

      #Start new process.
      $process = Start-Process $path

      #Get process id started.
        $process.Id
```

Chapter 4 Windows Management Instrumentation

Using WMI classes

One of the most striking features of PowerShell is its ability to interact with almost all settings of a machine. These settings are accessible through the WMI class. In these WMI examples, I will only use Win32 classes. However, in order to access my machine configurations, there is an endless list of them. A list of all of the classes is available here.

The WMI is an infrastructure management data and operations for Windows operating systems. It not only works with PowerShell, but also other languages such as C/C++, VB, and most other scripting languages built for Windows systems.

As I previously mentioned, I will only use Win32 classes in my example. These classes provide a large number of machine interactions such as getting the disk space available or getting our RAM memory usage. In the following table, you can see the main Win32 classes groups.

Table 6: Win32 Classes Groups

Class Group	Definition
Computer System Hardware Classes	Hardware-related objects.
Installed Applications Classes	Software-related objects.
Operating System Classes	Operating system-related objects.
Performance Counter Classes	Raw and calculated performance data from performance counters.
Security Descriptor Helper Class	Class that provides methods to convert security descriptors between different formats.
WMI Service Management Classes	Management for WMI.

Access WMI Classes

To use the WMI class in PowerShell, you need to use the command `Get-WmiObject` followed by the class name. The following code block shows an example of a WMI class invocation.

```
Get-WmiObject Win32_DiskDrive
```

Calling this command will result in a set of information regarding your installed disks and its partitions.

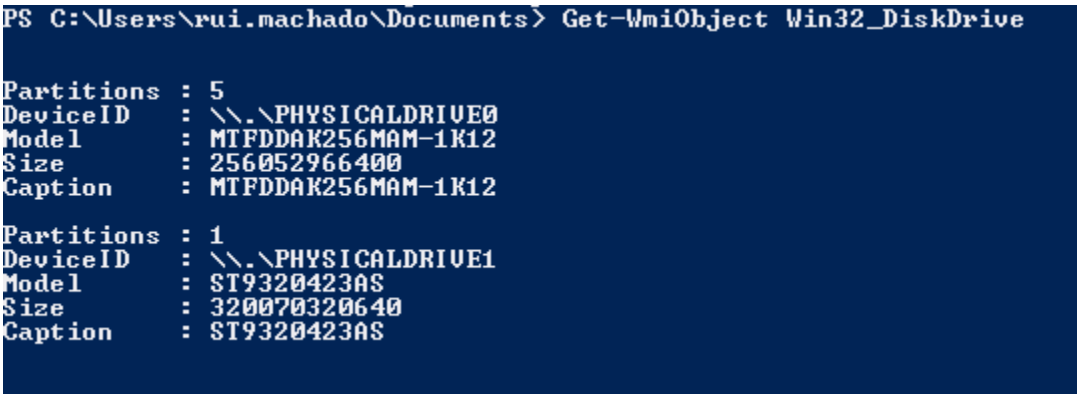


Figure 38: Calling Get-WmiObject Win32_DiskDrive

WMI has an additional feature that is usually not explored by users called WMI´s WQL language, which allows you to query any WMI class. To use it, you must reference the `WmiSearcher` object. The following code block shows you how you can use it in your PowerShell scripts:

```
#Instantiate a WmiSearcher object with the query you wish to run.
$query = [WmiSearcher]"select * from Win32_DiskDrive where
Size>300000000000"

#Run the query to retrieve the result.
$query.Get()
```

The previous script retrieves all disks in your machine that have a size greater than a predefined size. If you run it in the interactive shell, the result will resemble Figure 39.

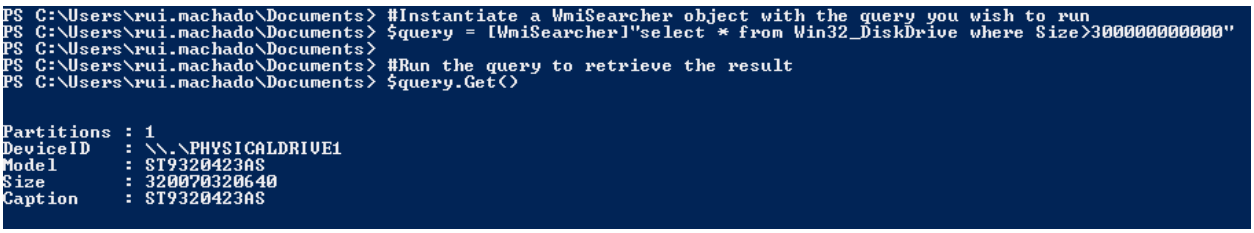


Figure 39: Run WQL queries

Exercise: Get Available Disk Space

For this challenge, we will setup a system administration scenario. One of the tasks of a professional in this area is to monitor the disk space of the various servers and make decisions and actions to avoid disk space allocation problems.

Performing this task manually is mundane and requires constant monitoring in order to avoid problems, so it is an ideal task to automate. In this first tutorial, we will create a simple script that lets you monitor available disk space. What I always try to encourage is establishing a scheduled task that runs every day, so that you won´t need to execute the script manually. Setting up a scheduled task that runs every day at 8am and saves the result to a log file, or even send you an alert email when disk space is running low, will give you free time to think about real problems instead of wasting time with monitoring and other routine tasks.

First create the basic routine, the function that will provide the information about a single machine. The following code block shows you how to create a function to retrieve this machine information.

```
function GetDiskInfo($serverName){

Get-WMIObject -ComputerName $serverName Win32_LogicalDisk |
       ?{($_.DriveType -eq 3)}|
       #Select which attribute to show.
             select @{n='Computer' ;e={"{0:n0}" -f ($serverName)}},
                   @{n='Drive' ;e={"{0:n0}" -f ($_.name)}},
                   @{n='Capacity (Gb)' ;e={"{0:n2}" -f ($_.size/1gb)}},
                   @{n='Free Space (Gb)';e={"{0:n2}" -f
($_.freespace/1gb)}},
                   @{n='Percentage Free';e={"{0:n2}%" -f
($_.freespace/$_.size*100)}}

}
```

Now that we have the main function, we just need to create a list of all of the servers in our domain that we need to monitor, and then iterate through each one and call our `GetDiskInfo` function.

In the following code block, you have the full script to list all of your servers' disk space with formatted values.

```
function GetDiskInfo($serverName){

Get-WMIObject -ComputerName $serverName Win32_LogicalDisk |
       ?{($_.DriveType -eq 3)}|
       #Select which attribute to show.
             select @{n='Computer' ;e={"{0:n0}" -f ($serverName)}},
                   @{n='Drive' ;e={"{0:n0}" -f ($_.name)}},
                   @{n='Capacity (Gb)' ;e={"{0:n2}" -f ($_.size/1gb)}},
                   @{n='Free Space (Gb)';e={"{0:n2}" -f
($_.freespace/1gb)}},
```

```
                    @{n='Percentage Free';e={"{0:n2}%" -f
($_.freespace/$_.size*100)}}

}

#List of servers to monitor.
$allServers = "ruimachado","computer2","computer3"

#Iterate each server.
$allServers | %{
      GetDiskInfo -serverName $_
}
```

The result of this script execution is a list of all disks with the respective free space for each of the servers declared in the array.

```
Computer          : ruimachado
Drive             : C:
Capacity (Gb)     : 222,23
Free Space (Gb)   : 65,89
Percentage Free   : 29,65%

Computer          : ruimachado
Drive             : D:
Capacity (Gb)     : 13,04
Free Space (Gb)   : 2,08
Percentage Free   : 15,93%

Computer          : ruimachado
Drive             : E:
Capacity (Gb)     : 2,00
Free Space (Gb)   : 2,00
Percentage Free   : 100,00%

Computer          : ruimachado
Drive             : F:
Capacity (Gb)     : 298,09
Free Space (Gb)   : 217,42
Percentage Free   : 72,94%
```

Figure 40: Disk Available List

You might not like how the information is being shown, and you might prefer a text file containing all of this information. To create this, you can store the information retrieved by the `GetDiskInfo` call inside of a foreach loop and then export it as a file.

```
#GetDiskInfo function goes here.

$allinfoArray=@()

#Iterate each server.
$allServers | %{
      $allinfoArray += GetDiskInfo -serverName $_
```

```
}

$allinfoArray | Format-Table | Out-File -FilePath "c:\temp\diskLog.txt"
```

This will generate a formatted table inside of a text file, as shown in Figure 41.

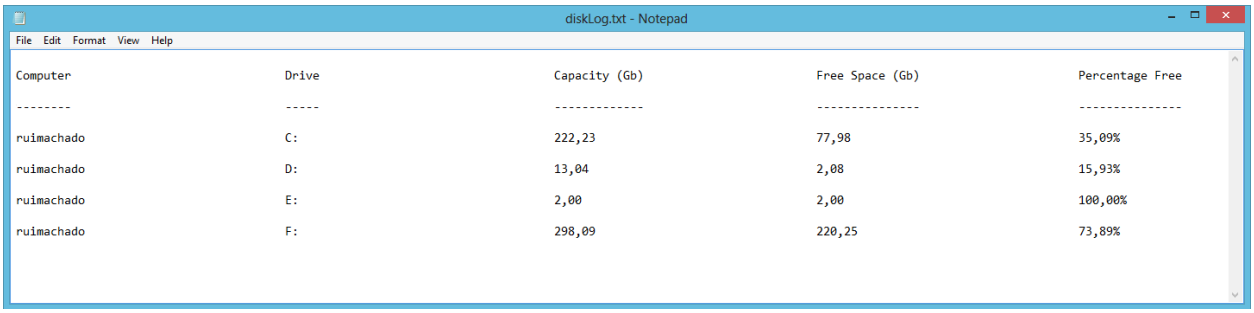
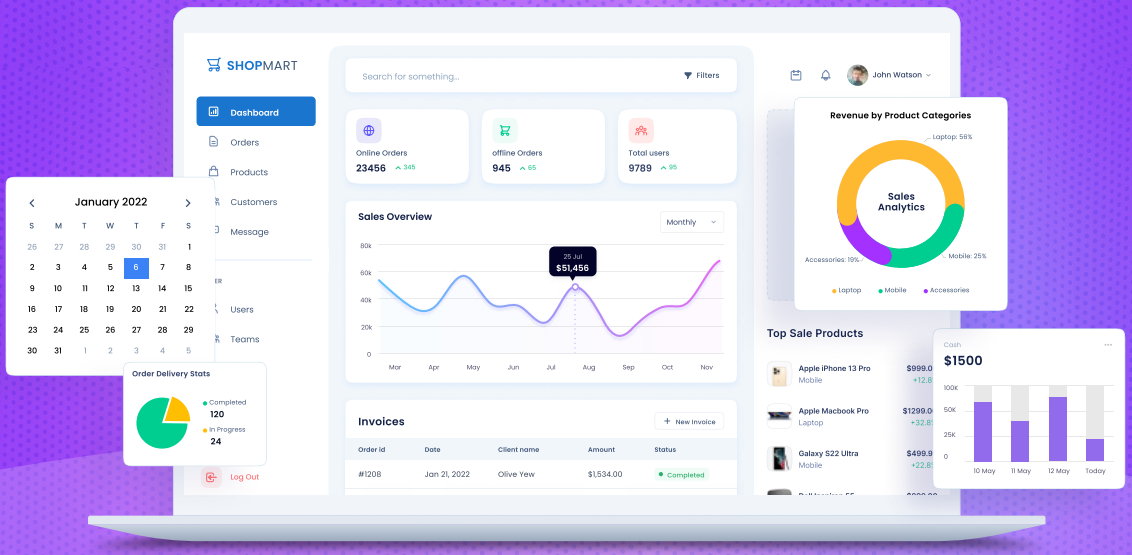


Figure 41: Formatted File with Disk Space

4.6 out of 5 stars

THE WORLD'S BEST UI COMPONENT SUITE FOR BUILDING POWERFUL APPS



GET YOUR FREE .NET AND JAVASCRIPT UI COMPONENTS

syncfusion.com/communitylicense

- 1,700+ components for mobile, web, and desktop platforms
- Support within 24 hours on all business days
- Uncompromising quality
- Hassle-free licensing
- 28000+ customers
- 20+ years in business

Trusted by the world's leading companies

Syncfusion

Chapter 5 Remote PowerShell

Using Remote PowerShell

Remotely accessing a local machine or running scripts on remote machines is a feature that Microsoft was reluctant to support, and until the first version of PowerShell there was no way to do it. Since PowerShell 2.0, things have changed and Microsoft has made a big effort to implement the Windows Management Foundation, which combines PowerShell with Windows Remote Management, offering tools for system administrators who can now manage all machines on one or more domains.

One thing that is important to note is that not all PowerShell commands allow remote access to computers. However, there are many ways to run scripts remotely, either by executing them on the remote computer or running them from your own computer.

Another important point before we move on to the practical cases is that some commands require that PowerShell be installed on both machines, while others may be run on machines without PowerShell installed.

In order to use PowerShell remotely on a computer, it must be active in that computer. To do so, use the `Enable-PsRemoting` command.

```
Enable-PsRemoting
```

Make sure you run the PowerShell console in administrator mode, otherwise you will see the error shown in Figure 42.

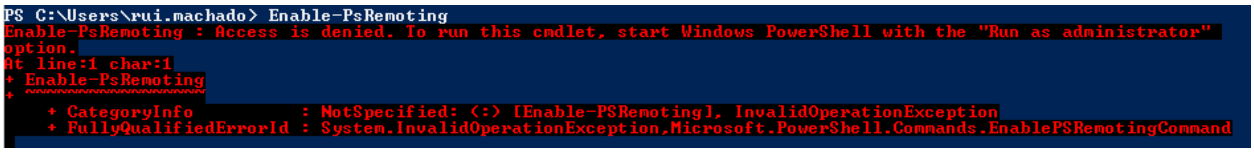


Figure 42: Enable-PSRemoting Error

Another error you might face is a network connection type. In order to activate remote PowerShell, all of your network connections types must be either private or domain. The error is shown in Figure 43.

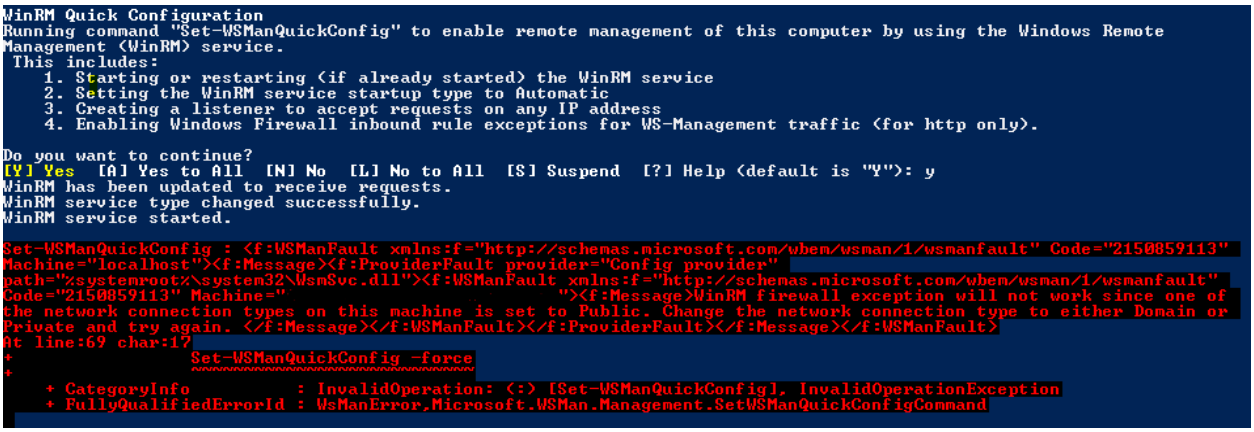


Figure 43: Enable-PSRemoting Network Connection Type

The solution to this problem is to change the connection type as previously mentioned, to either public or domain. To do so, you must invoke the following function, which will change the network type to the one specified by the type parameter. You can invoke this script for just one network, or activate the flag (Boolean parameter “All”) to change it for all of your networks.

```
function ChangeNetworkConType{
	param(
		[ValidateSet("Domain","Public","Private")]
		[string][Parameter(Mandatory=$true)]$Type,
		[string][Parameter(Mandatory=$false)]$NetworkName,
		[bool][Parameter(Mandatory=$true)]$ChangeAll
	)
	#Instantiate a network manager object using its CLSID (CLSID is a com object ID).
	[Activator]::CreateInstance([Type]::GetTypeFromCLSID([Guid]"{DCB00C01-570F-4A9B-8D69-199FDBA5723B}")) | %{

		#Get all connections.
		$allConns = @()
		if($ChangeAll -eq $true){
			$_.GetNetworkConnections() | %{ $allConns+= $_.GetNetwork()}
		}else{
			$_.GetNetworkConnections() | ?{$_.GetNetwork().GetName() -like "*$NetworkName*"} | %{$allConns+=$_.GetNetwork()}
		}

		$allConns | %{
			<#
				Network Connection Types Available
				Public  - 0
				Private - 1
				Domain  - 2 //Won´t cover it in this script
```

```
                  #>
                  #Because we will change the context of execution to a
switch, we must save the network connection to a variable.
                  $networkConnection = $_

                  switch($Type){
                        "Public"
                        {
                              $networkConnection.SetCategory(0);
                              break;
                        }
                        "Private"
                        {
                              $networkConnection.SetCategory(1);
                              break;
                        }
                        default {"Only Private or Public type is
implemented."}
                  }
            }
      }
}
```

To invoke the command, just run the following code block:

```
ChangeNetworkConType -Type "Private" -ChangeAll $true
```

After you overcome all of these obstacles, enabling PowerShell remote shouldn´t give you any more problems, but you may need to authorize several changes to your machine.

```
PS C:\Windows\system32> Enable-PSRemoting

WinRM Quick Configuration
Running command "Set-WSManQuickConfig" to enable remote management of this computer by using the Windows Remote
Management (WinRM) service.
 This includes:
    1. Starting or restarting (if already started) the WinRM service
    2. Setting the WinRM service startup type to Automatic
    3. Creating a listener to accept requests on any IP address
    4. Enabling Windows Firewall inbound rule exceptions for WS-Management traffic (for http only).

Do you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
WinRM is already set up to receive requests on this computer.
WinRM has been updated for remote management.
Created a WinRM listener on HTTP://* to accept WS-Man requests to any IP on this machine.
WinRM firewall exception enabled.


Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name: microsoft.powershell SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run
Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name: microsoft.powershell.workflow SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run
Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y

Confirm
Are you sure you want to perform this action?
Performing operation "Set-PSSessionConfiguration" on Target "Name: microsoft.powershell32 SDDL:
O:NSG:BAD:P(A;;GA;;;BA)(A;;GA;;;RM)S:P(AU;FA;GA;;;WD)(AU;SA;GXGW;;;WD). This will allow selected users to remotely run
Windows PowerShell commands on this computer".
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"): y
PS C:\Windows\system32>
```

Figure 44: Enable-PSRemoting run with no errors

Identify Remote PowerShell Compatible Commands

While working with remote PowerShell commands, you might face two types of context application: one in which you have a remote compatible command and you invoke it remotely, and another case in which you don´t have a remote compatible command. In this situation, you need to invoke the script or command directly on the other machine through a remote session. To evaluate which of these two scenarios are available for you, you need to check if a certain command is compatible with remote PowerShell.

To evaluate a command's compatibility in this context, you can simply check if it has a computer name parameter, which means that you can get a full compatible list just by using the `Get-Command` command. This will return a list of all commands available in PowerShell and then filter that list to show only those that contain a computer name parameter.

```
#Get all commands.
Get-Command -CommandType Cmdlet | %{
      #Filter the list to show those that contain computer name parameter.
      if(($_.Parameters -ne $null) `
            -and ($_.Parameters["ComputerName"] -ne $null)){
            #Return the command name.
            $_.Name
      }
}
```

Note: To invoke a command on a remote machine, it might need to have PowerShell installed; however, there is a common technique used to overcome that problem, which is to establish a remote desktop session.

Test a Remote Connection

Testing the connectivity between two computers is allowed in PowerShell through its `Test-Connection` command, which works like the old ping command. It sends Internet Control Message Protocol (ICMP) echo request packets (pings) to one or more remote computers and returns the echo response replies.

```
#Test connection between this computer and another
Test-Connection -ComputerName "WIN-FI6G73MN5BB"
```

This command is easy to call in its most basic form; just call it with the target computer name parameter and that's it. The result of this call is shown in Figure 45, which in this case is a success.

```
PS C:\Users\rui.machado> #Test connection between this computer and another
PS C:\Users\rui.machado> Test-Connection -ComputerName "WIN-FI6G73MN5BB"

Source        Destination     IPV4Address      IPV6Address                              Bytes    Time(ms)
------        -----------     -----------      -----------                              -----    --------
RUIMACHADO    WIN-FI6G73MN5BB 192.168.45.129                                            32       3
RUIMACHADO    WIN-FI6G73MN5BB 192.168.45.129                                            32       3
RUIMACHADO    WIN-FI6G73MN5BB 192.168.45.129                                            32       0
RUIMACHADO    WIN-FI6G73MN5BB 192.168.45.129                                            32       0
```

Figure 45: Test connection

After you check that the connection is established, you are ready to start invoking commands and scripts on that machine.

Invoke Scripts in Remote Machines

Although the remote connection connects to a single computer, the host application can run the `Invoke-Command` to run commands on other computers. With this concept, you are able to use the `Invoke-Command` command, which runs commands on a local or remote computer and returns all output from the commands, including errors, just by specifying the computer name as a parameter. A temporary connection is automatically created between your computer and the target machine.

```
#Invoke the command with the target computer and the script block
Invoke-Command -ComputerName "WIN-FI6G73MN5BB" -ScriptBlock {Get-Process}
```

When invoking this command, you might be faced with the following error, which can be caused by several situations. I will give you the solution to most common one.

```
[WIN-FI6G73MN5BB] Connecting to remote server WIN-FI6G73MN5BB failed with the
following error message : WinRM cannot process the request. The following
error with errorcode 0x80090311 occurred while using Kerberos authentication:
There are currently no logon servers available to service the logon request.
 Possible causes are:
  -The user name or password specified are invalid.
  -Kerberos is used when no authentication method and no user name are
specified.
  -Kerberos accepts domain user names, but not local user names.

  -The Service Principal Name (SPN) for the remote computer name and port
does not exist.
  -The client and remote computers are in different domains and there is no
trust between the two domains.
 After checking for the above issues, try the following:
  -Check the Event Viewer for events related to authentication.
  -Change the authentication method; add the destination computer to the
WinRM TrustedHosts configuration setting or use HTTPS transport.
 Note that computers in the TrustedHosts list might not be authenticated.
```

To solve this problem, assuming that you have already enabled remote PowerShell on both machines using the `Enable-PsRemoting` command, you can establish a trust relation when using WinRM between both machines by using the following code. You must run this with administrator rights on both the destination and target machines.

```
Set-Item wsman:\localhost\client\trustedhosts “DestinationMachineName”
```

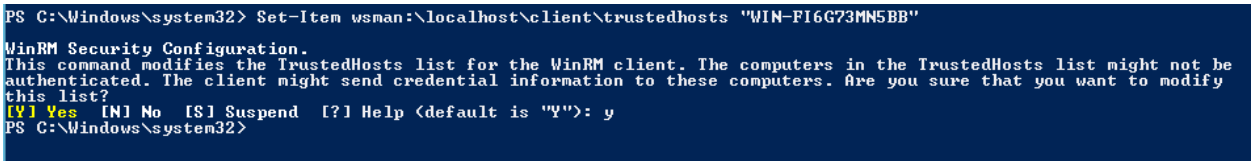


Figure 46: Set Trusted Hosts

Note: Remote connection between machines in PowerShell is managed by a security and authentication mechanism named Kerberos. This mechanism prevents the remote computer you are connected to from using your account to connect to additional computers.

Chapter 6 Structured Files

Manipulating XML Files

Import XML from File

To import an XML structure from a previously built XML file, you can use the `Import-Clixml` command if your XML file was created by an `Export-CliXml` command. CLIXML in PowerShell is used to store object and its representation object to a file.

To manipulate your own XML structures, you should use `Get-Content` command, casting the object to XML. In this example, because my XML file was created outside PowerShell, I will use the `Get-Content` command.

```
$xmlFile = [xml](Get-Content -Path "C:\Users\rui.machado\Desktop\X.xml")

#Creates a DOM Object available
$xmlFile.PersonList | %{
      $_
}
```

Load XML File from String

PowerShell allows you to easily build a DOM object of an XML structure using the .NET object `System.Xml.XmlDocument` and invoking the `LoadXml($xmlString)`, building a DOM object and allowing you to iterate in your XML tree structure. Now to try it, build a script to load an XML file from a string. The following code block shows the XML string:

```
<PersonList>
      <Person id='1'>
            <Name>Rui</Name>
            <Age>24</Age>
            <Address>Street A</Address>
      </Person>
      <Person id='2'>
            <Name>Peter</Name>
            <Age>45</Age>
            <Address>Street B</Address>
      </Person>
      <Person id='3'>
            <Name>Mary</Name>
            <Age>10</Age>
            <Address>Street C</Address>
      </Person>
```

```
</PersonList>
```

Using the XML file you are going to work with, create a routine to process the requirements.

```
$xmlString = @"
      <PersonList>
            <Person id='1'>
                  <Name>Rui</Name>
                  <Age>24</Age>
                  <Address>Street A</Address>
            </Person>
            <Person id='2'>
                  <Name>Peter</Name>
                  <Age>45</Age>
                  <Address>Street B</Address>
            </Person>
            <Person id='3'>
                  <Name>Mary</Name>
                  <Age>10</Age>
                  <Address>Street C</Address>
            </Person>
      </PersonList>
"@

$xmlDoc = new-object System.Xml.XmlDocument

$xmlDoc.LoadXml($xmlString)

$xmlDoc.PersonList | %{
      write $_.Person.Name
}
```

Export XML to File

To export a data set to an XML file, you have two options. First, you can use the **Export-CliXml** command. However, this is a very specialized XML format defined by PowerShell that creates an XML-based representation of an object or objects and stores it in a file. You can then use the Import-CLIXML command to recreate the saved object based on the contents of that file. The second option is to export a structure using the .NET System.Xml.XmlDocument object, which allows you to save it to the file system. You can also pipe the output to the **Out-File** command, which gives you more control over how the file is created. This second approach creates XML files that are more easily consumed outside of PowerShell scripts.

```
#The structure to export.
$xml = @"
<CarsList>
      <Car>
            <Brand>Ferrari</Brand>
```

```
      </Car>
      <Car>
            <Brand>Porsche</Brand>
      </Car>
</CarsList>
"@

#Create a new XML document.
$xmlDoc = New-Object System.Xml.XmlDocument

#Load the xml structure.
$xmlDoc.LoadXml($xml)

#Define the file path.
$pathToFile="c:\temp\FileXML.xml"

#Save the structure to a file.
$xmlDoc.Save($pathToFile)
```

In the previous example, I used the .NET object `System.Xml.XmlDocument` to load and save the XML file to file system; however, you can use the out file command as well, as you can see in the following example.

```
#The structure to export.
$xml = @"
<CarsList>
      <Car>
            <Brand>Ferrari</Brand>
      </Car>
      <Car>
            <Brand>Porsche</Brand>
      </Car>
</CarsList>
"@

#Define the file path.
$pathToFile="c:\temp\FileXML2.xml"

#Export the file.
$xml | Out-File $pathToFile
```

Both of these previous scripts will result in the same XML file. The difference is just in the amount of code created and cleanliness of it, which in the second case is much better. The resulting structure is shown in the following sample.

```
<CarsList>
      <Car>
            <Brand>Ferrari</Brand>
      </Car>
```

```
      <Car>
            <Brand>Porsche</Brand>
      </Car>
</CarsList>
```

Manipulating CSV Files

Import CSV from File

Importing CSV files in PowerShell might be important for projects like a technology migration in which its records will be inserted in a SQL Server database. In the following code block, we have a small example of a CSV file, which in this case will be imported to PowerShell as a PSObject, allowing you to iterate it as you wish.

```
Rui;24;Portugal
Tony;45;USA
Thiago;12;Brazil
Anna;56;Germany
```

To import a CSV file, use the `Import-CSV` command, having as parameters the path to the CSV file, the file delimiter, and the header, which identifies every column in the structured file. If your file has headers, you don´t need to use this parameter.

```
#Path to CSV file.
$path="c:\temp\ListaUsers.csv"

#Import your CVS file defining the file path, CSV delimiter and the header.
$csv=Import-CSV -Path $path -Delimiter ";" -Header
"Name","Age","Nationality"

#Storing the CSV to a variable allows you to manipulate its content.
$csv | %{
      $_.Name
      $_.Age
      $_.Nationality
}
```

Export CSV to File

Exporting to CSV in PowerShell is a little bit more complicated than any other export for one simple reason: PowerShell's `Export-Csv` command works with PSObjects, so you need to convert your structures to one of this kind before exporting it to CSV. In the following code block, I will be showing you how to create a PSObject from an array, and then export it to CSV. To export, I will be using the `Export-Csv` command, which converts objects into a series of comma-separated (CSV) strings and saves the strings into a file.

PSObject

Encapsulates a base object of type Object or type PSCustomObject to allow for a consistent view of any object within the Windows PowerShell environment.

```
#Path to export.
$path = "c:\temp\testeExport.csv"

#Structure to export.
$csv=("Rui",24,"Portugal"),("Helder",29,"China"),("Vanessa",24,"Brasil")

#Initialize the array that will be exported.
$allRecords = @()

$csv | %{
      #Export-CSV separates columns from PSObject members, so we need to
create one.
      $customCSV = New-Object PSObject

      #Add the name member to the PSObject
      $customCSV  | Add-Member -MemberType NoteProperty -Value $_[0] -Name
"Name"

      #Add the name member to the PSObject
      $customCSV  | Add-Member -MemberType NoteProperty -Value $_[1] -Name
"Age"

      #Add the name member to the PSObject
      $customCSV  | Add-Member -MemberType NoteProperty -Value $_[2] -Name
"Nationality"

      #Add the PSObject to the array that stores every object to export.
      $allRecords+=$customCSV
}

#Export as CSV.
$allRecords | Export-Csv -Path $path -NoTypeInformation -Delimiter ";"
```

Load CSV and Send Email

```
Rui,ruimachado@live.com.pt,Hello
Machado,ruimachado@outlook.pt,Hi!
```

With the previous common separated values file, we´ll use its values and send an email to every recipient defined.

```
<#
      Routine that matter CSV file. As the CSV file has columns use the
    Header-parameter to set the column headers. We also need to define the
delimiter of the CSV,
      which in this case is a comma, but can be any other.
#>

$userList= Import-Csv -Path "C:\Users\rui.machado\Desktop\Users.csv"`
                              -Header "Name", "Email", "Subject"`
                              -Delimiter ","`

#SMTP Server
$smtpServer="smtp.gmail.com"

#Create a new .NET SMTP client onject (587 is the default smtp port)
$smtpClient = new-object Net.Mail.SmtpClient($smtpServer,587)

      #Create a new .NET Credential object.
      $credentials = New-Object Net.NetworkCredential

      #Set your credentials username.
      $credentials.UserName="powershellpt@gmail.com"

      #Set your credentials password.
      $credentials.Password="rpsrm@89"

#Set the smtp client credential with the one created previously.
$smtpClient.Credentials=$credentials

      #Create a new .NET mail message object.
      $email = new-object Net.Mail.MailMessage

            #Initialize the from field.
            $from="powershellpt@gmail.com"

            #Initialize the mail message body.
            $message="Este é um email do powershellpt. Seja bem vindo "

      #Set the from field in the mail message.
      $email.From = $from
```

```
      #Set the Reply to field in the mail message.
      $email.ReplyTo = $from

      #Set the body field in the mail message.
      $email.body = $message

# foreach user in the CSV file, send the email.
$userList | %{
      #Just send to users with a defined email.
      if($_.Email -ne $null)
      {
            #Set recipient name.
            $nome=$_.Name

            #Add recipient email to the Mail message object.
            $email.To.Add($_.Email)

            #Set the subject.
            $email.subject = $_.Subject

            #This is a Google smtp server requirement.
            $smtpClient.EnableSsl=$true

            #Send the email.
            $smtpClient.Send($email)
      }
}
```

Manipulating TXT Files

Import TXT from File

PowerShell does not have an explicit import command. Instead, you need to use the Get-Content command to retrieve the text file content, which means that you only need to specify as a parameter the file path. After you invoke the command, PowerShell will instantiate a System.String object containing the text file content.

```
#Path to the txt file.
$path = "C:\temp\test.txt"

#Get content from the txt file.
$content = Get-Content -Path $path

#Print content.
$content
```

Export TXT to File

Exporting an object to a text file is provided in PowerShell through its `Out-File` command, which sends a specific output to a file as a string. This command also allows you to define as a parameter the encoding of the file you want to create.

```
#Path to the txt file.
$path = "C:\temp\newTest.txt"

<#
      EXAMPLE 2
      Create txt file and write to it.
#>

"Write to Txt" | Out-File $path

<#
      EXAMPLE 2
      Create txt file and write to it, setting the encoding to ACII.
#>
"Write to Txt" | Out-File $path -Encoding ASCII

<#
      EXAMPLE 3
      Create txt file and write to it, setting the encoding to ACII and the
width of 5,
      which means that only the first 5 characters will be exported.
#>
"Write to Txt" | Out-File $path -Encoding ASCII -Width 5
```

Using XSL to Transform XML Files

Sometimes you might need to transform XML files in order to respect a predefined XML schema, for instance a SQL Server database table. Imagine that you wish to query an SQL Server database of a client, return the dataset as a XML document, transform it using an XSL script, and then insert it in another SQL database table from another client. This is a very powerful tool for integration projects in which source and destination structures are not equal. In the following code block, I will you how you can use .NET `System.XML.XSL.XSLCompiledTransform` object to apply a XSL script to an XML file.

To input an XML file, use the following one, which is a list of people. The goal is to transform the three elements of a person (Name, Age, Address) into a comma-separated values element (Data).

```
<PersonList>
            <Person id='1'>
                  <Name>Rui</Name>
```

```
                  <Age>24</Age>
                  <Address>Street A</Address>
            </Person>
            <Person id='2'>
                  <Name>Peter</Name>
                  <Age>45</Age>
                  <Address>Street B</Address>
            </Person>
            <Person id='3'>
                  <Name>Mary</Name>
                  <Age>10</Age>
                  <Address>Street C</Address>
            </Person>
</PersonList>
```

The XSL file you are using is the following, which makes the transformation defined previously:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:template match="/">
            <PersonList>
                  <xsl:for-each select="PersonList/Person">
                        <Person>
                              <Data>
                                    <xsl:value-of select="Name" />;
                                    <xsl:value-of select="Age" />;
                                    <xsl:value-of select="Address" />
                              </Data>
                        </Person>
                  </xsl:for-each>
            </PersonList>
      </xsl:template>
</xsl:stylesheet>
```

To achieve a transformation with an XSL file in PowerShell, use the `System.XML.XSL.XSLCompiledTransform as` I have previously mentioned, particularly two of its methods, the Load() method.

The first one will load the XSL script from its path, and the Transform() method applies the transformation script to an input XML file and returns transformed structure to an output XML. Both of these XML files are defined by their path as parameters of this method.

```
#Path to XSL stylesheet
$xslStylesheetPath = "C:\temp\XsltFile.xsl"

#Path to XML Input file
$xmlInputPath = "C:\temp\InputXml.xml"

#Path to XML Ouput file. The result of the transformation.
```

```
$xmlOutputPath = "C:\temp\OutputXml.xml"

#Instantiate the XslCompiledTransform .NET object.
$xslt = New-Object System.Xml.Xsl.XslCompiledTransform;

#Load the XSL script.
$xslt.Load($xslStylesheetPath);

#Applies the transformation to the input XML and return the result as an
ouput XML.
$xslt.Transform($xmlInputPath, $xmlOutputPath);
```

As a result, the transformation resulted in the following XML structure, with the person elements transformed into one single comma-separated values element.

```
<?xml version="1.0" encoding="utf-8"?>
<PersonList>
      <Person>
            <Data>Rui;24;Street A</Data>
      </Person>
      <Person>
            <Data>Peter;45;Street B</Data>
      </Person>
      <Person>
            <Data>Mary;10;Street C</Data>
      </Person>
</PersonList>
```

Chapter 7 SQL Server and PowerShell

Using databases is a regular activity for most of today's developers, which means that modern programming languages must be able to connect to a wide variety of databases to be successful, and PowerShell is no exception. Although database connection and querying is always available out of the box thanks to the .NET framework, it is limited to its providers.

Even if you feel familiar with .NET, you should consider other options such as PowerShell modules, which bring much better solutions to optimize your interaction with SQL Server. In this chapter, you will learn to work with SQLPS, Microsoft´s PowerShell SQL Server official module. I use this module every day and it has proven to be very stable and efficient. To start using it, you need to install SQL Server or install some requested features.

Install SQLPS

If you have SQL Server installed on your machine, then the SQLPS module is installed along with that installation. However, you can simply install three stand-alone packages from the Microsoft SQL Server 2012 feature pack. Click here to download these packages, and click on **Install Instructions** to open a drop-down list of all the features available. Download and install the following in this order:

1. Microsoft System CLR Types for Microsoft SQL Server 2012 (SQLSysClrTypes.msi)
2. Microsoft SQL Server 2012 Shared Management Objects (SharedManagementObjects.msi)
3. Microsoft Windows PowerShell Extensions for Microsoft SQL Server 2012 (PowerShellTools.msi)

After you finish installing these packages, you will be able to load the SQLPS module and start working with your SQL Databases directly in PowerShell. An important notice is to select the correct version of the install package according to your processor architecture. In Figure 47, you see an example of the choices you will have for each feature to install.

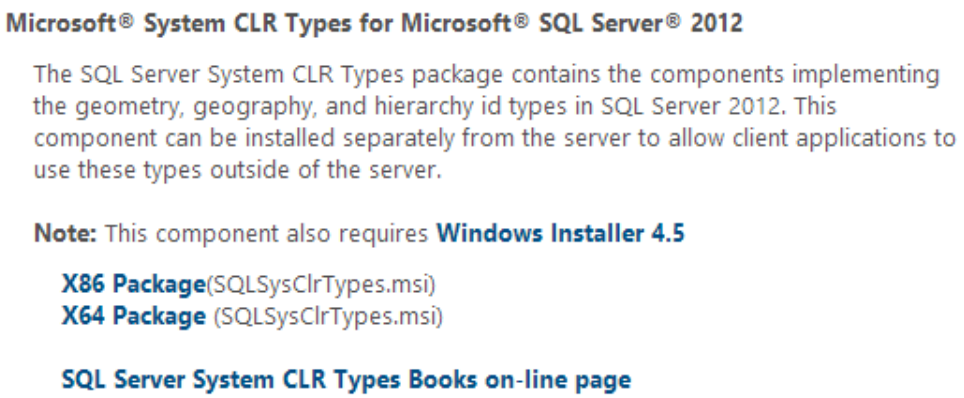


Figure 47: Install SQLPS feature

Add SQL Snap-in

To add the SQLPS module to a script, you just need to invoke the `Import-Module` command, which will add the module to the current session. Note that the modules you import must be installed on the local computer or a remote computer if you wish to run these scripts on it.

```
#Import the SQLPS module
Import-Module “sqlps”
```

After importing the SQLPS module to your script, some new commands should be available in PowerShell. As you can see in Figure 48, you now have the ability to execute a SQL command, invoke a process on a cube, or even process a dimension on a cube from Analysis Services.

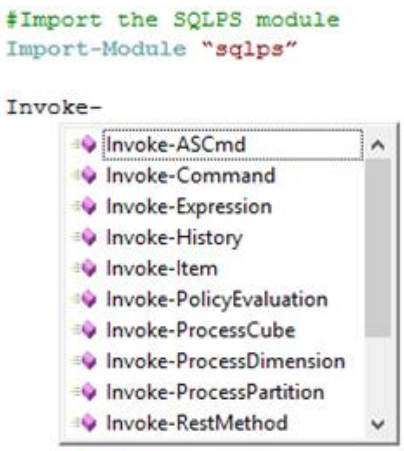


Figure 48: New SQL commands

Invoke SQL Query

```
#Import the SQLPS module only if it is not already loaded.
if(-not (Get-Module "sqlps")){
      Import-Module “sqlps”
}
#Set the database.
$database="rmBlog"

#Set the server and instance.
$server = ".\PRI01"

#read the post name from the interactive shell.
"Please enter a post to seach`n"
```

```
$post = Read-Host

#Set the query.
$query = "SELECT p.PostTitle,
         p.PostText,
         p.PostDate
FROM Post p
WHERE PostTitle like '%$post%'"

$query
#Invoke the command providing as parameters all that are necessary.
Invoke-Sqlcmd -ServerInstance $server -Database $database -Query $query
```

Invoking this script will return a dataset of results directly from SQL Server that you can use in the rest of your script. Imagine you want the result as an XML file. Several possible solutions might come to mind that would be acceptable, but if you are using SQL Server, why not use it to retrieve our result as an XML file? You just need to change the query in order to convert the result and export it to a XML file.

```
#...(Some code omitted)

#Set the query.
$query = “SELECT p.PostTitle,
         p.PostText,
         p.PostDate
FROM Post p
WHERE PostTitle like ‘%$post%’
FOR XML AUTO, ROOT(‘MyPosts’)”

#Invoke the command providing as parameters all that are necessary.
$result = Invoke-Sqlcmd –ServerInstance $server –Database $database –Query
$query

#Load the result as XML.
$xmlDoc = new-object System.Xml.XmlDocument
$xmlDoc.LoadXml($result[0])

#Export as XML.
$xmlDoc.InnerXml | Out-File “c:\temp\testeSQL.xml”
```

After you invoke this script, you will have an XML file with the full DOM object available for you to work with. The following code block shows you the result of running this script.

```
<MyPosts>
    <p PostTitle="Snnipet Creator C#"
       PostText="My First Post"
       PostDate="2013-06-18T20:29:18.857" />
</MyPosts>
```

There is one final important warning about using this SQL module. When you run the script to invoke a call to a SQL database, you might face the following error:

```
Invoke-Sqlcmd : A network-related or instance-specific error occurred while
establishing a connection to SQL Server. The server was not found or was not
accessible. Verify that the instance name is correct and that SQL Server is
configured to allow remote connections. (provider: SQL Network Interfaces,
error: 26 - Error Locating Server/Instance Specified)
```

If all server and database parameters are correct, this might be related to the SQL Server configuration, which you need to change in order to allow remote connections for that server. You can use SQL Server Management Studio as you can see in Figure 49.

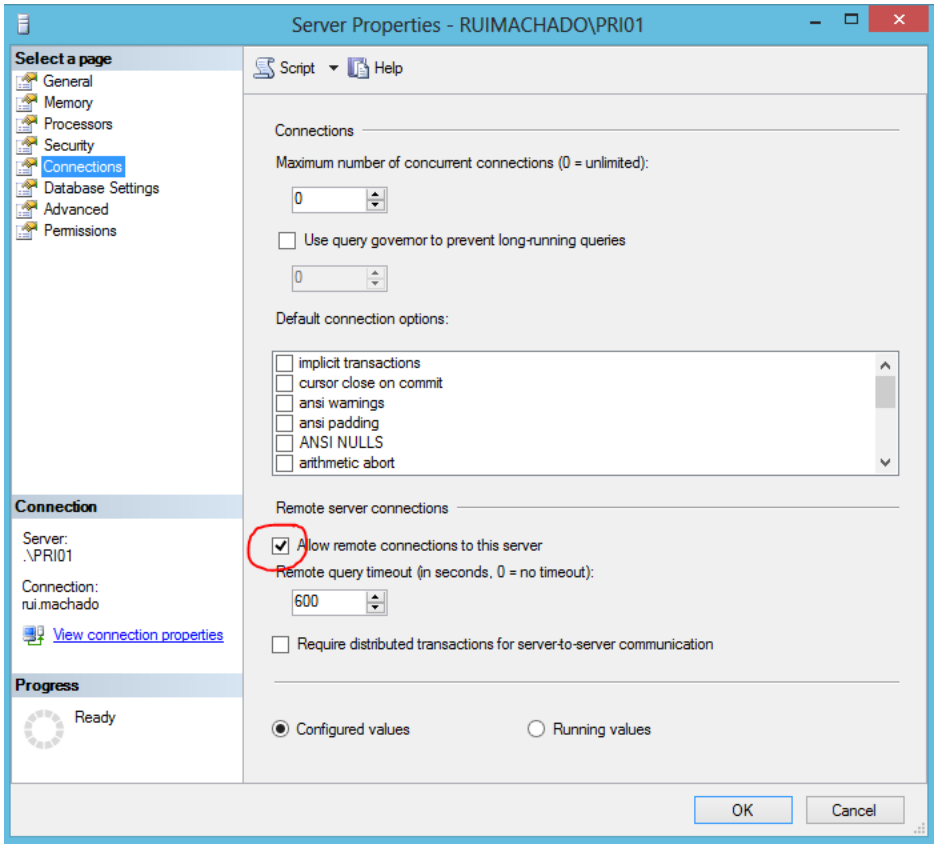


Figure 49: Allow remote connections

Chapter 8 Microsoft Office Interop

Microsoft named its set of interaction capabilities with Microsoft Office as Primary Interop Assemblies (PIAs), a package that contains all the major assemblies of the various Microsoft Office Suite products. These packages allow you to interact with objects created by the Office products. For example, if you create an Excel file with two sheets, through the Interop assemblies you can get these objects and iterating with them. To use it, you just need to install the PIAs assemblies and have Microsoft Office installed on your computer.

Note: Download PIAs assemblies here.

A scenario that can portray the advantages of using Interop is simply trying to answer this question: "How many sheets of your Excel file contain links to SQL databases?"

It is possible to manually answer this question just using your Excel files, but if you have two hundred Excel files, how long do you think that will take? I'll show you how to quickly make a PowerShell script that runs through all these files and returns a CSV file with all this information processed and available in a tabular list. Answering these kinds of questions can be important in analysis projects in which you need to identify dependencies between artifacts of your project files.

Although my exercise will be about Excel files (workbooks and sheets), PIAs allow you to work with almost all Office products. The following is a complete list of allowed products:

- Microsoft Access
- Microsoft Excel
- Microsoft InfoPath
- Microsoft Outlook
- Microsoft PowerPoint
- Microsoft Project
- Microsoft Publisher
- Microsoft SharePoint Designer
- Microsoft Visio
- Microsoft Word

Using PIAs Assemblies

Using any PIAs assembly always has the same algorithm. You start by loading the assembly to your script and then you create the primary Office file type object and use it as you wish.

To load an assembly in PowerShell, you can use the same method often used in .NET projects, which is LoadWithPartionName from the Reflection.Assembly namespace.

```
[Reflection.Assembly]::LoadWithPartialname("AssemblyName")
```

To load an Office assembly, you have the following options:

```
#Load Microsoft Office Excel Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Excel")

#Load Microsoft Office Access Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Access")

#Load Microsoft Office InfoPath Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.InfoPath")

#Load Microsoft Office OneNote Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.OneNote")

#Load Microsoft Office Outlook Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Outlook")

#Load Microsoft Office PowerPoint Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.PowerPoint")

#Load Microsoft Office Publisher Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Publisher")

#Load Microsoft Office Visio Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Visio")

#Load Microsoft Office Word Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Word")
```

Using PIAs is similar from assembly to assembly, so I will only show you how to use the Excel assembly for simple scenarios. You can use Word, Access, or any other as a similar script structure.

Create an Instance of an Excel Application

To create an instance of an Excel application, you should use the ApplicationClass within the PIAs Excel assembly. Note that PIAs always use ApplicationClass in every assembly to instantiate a new application, in this case Excel. This is like opening Excel through an executable file; PowerShell will create that Excel process as well.

```
$excel = new-object Microsoft.Office.Interop.Excel.ApplicationClass
```

Tip: Because PowerShell creates a new Excel process and the only way to close it is killing the process, its a good idea to kill every Excel process before creating the Excel application; to do so, run the following command: `Get-Process | where{$_.ProcessName -like "*EXCEL*"} | kill`.

```
#TIP: Kill every Excel application before creating a new one.
Get-Process | where{$_.ProcessName -like "*EXCEL*"} | kill

$excel = new-object Microsoft.Office.Interop.Excel.ApplicationClass
```

Retrieve Data from Excel File

After loading the assembly and creating your Excel application object, you are ready to start pulling data out of it. For this example, we will use an Excel 2013 worksheet with a table in it and we will try to get that data.

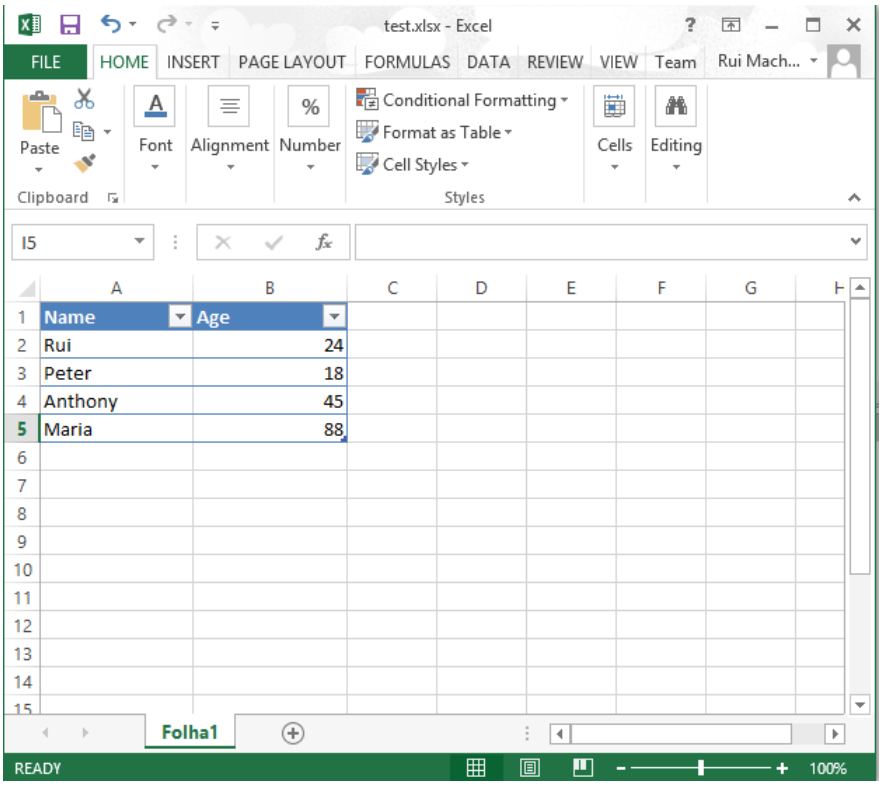


Figure 50: Excel File

To build our script, we need all previous knowledge about PIAs assemblies, as we will initialize a new Excel application, load that specific file shown in the previous image, and query the sheet for its table data. The following example is just a way of doing a retrieval of an Excel workbook; the main goal is to learn some PIAs artifacts usage.

```
#Load Excel PIAs assembly.
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Excel")

#Kill all Excel processes.
Get-Process | where{$_.ProcessName -like "*EXCEL*"} | kill

#Instantiate a new Excel application
$excel = new-object Microsoft.Office.Interop.Excel.ApplicationClass

#Path to our Excel file.
$filePath="c:\temp\test.xlsx"

#Instatiate a new workbook and then its path.
$book = $excel.Workbooks.Open("$filePath")

#Get the sheet - In this case we just have one.
$sheet = $book.Worksheets.Item(1)

#Get all list objects.
$lObjects = $sheet.ListObjects

#Get the range of data.
$range = $lObjects.Item(1).Range

#Print all cell data.
$lObjects.Item(1).Range.Rows | %{
      $row = $_.Row
      $lObjects.Item(1).Range.Columns | % {
            $col = $_.Column

            #Cell item
            write $range.Item($row,$col).Value2
      }
}
```

The result should be displayed as in Figure 51.

```
Name
Age
Rui
24
Peter
18
Anthony
45
Maria
88
```

Figure 51: Result of an Excel Data Retrieval

Exercise: How Many SQL Server Connections are in That Excel File?

In this exercise, I will answer that first challenge I mentioned in the beginning of this chapter, how to identify how many SQL Server connections in an Excel workbook. I will retrieve the number of connections; however, you can increase the complexity of this exercise and also retrieve the connection string of that SQL Server connection and even the range in which it is used.

```
#Load Excel PIAs Assembly
[Reflection.Assembly]::LoadWithPartialname("Microsoft.Office.Interop.Excel")

#Kill all Excel processes.
Get-Process | where{$_.ProcessName -like "*EXCEL*"} | kill

#Instantiate a new Excel application.
$excel = new-object Microsoft.Office.Interop.Excel.ApplicationClass

#Path to our Excel file.
$filePath="c:\temp\test.xlsx"

#Instatiate a new workbook and then its path.
$book = $excel.Workbooks.Open("$filePath")

#Get all book connections.
$bookConnections = $book.Connections

#Get total number of connections.
$totalConnections = $bookConnections.Count

#Initialize the counter of SQL connections.
$conSql = 0

#Foreach connection check if it is SQL Server.
$bookConnections | %{
      if($_.Type -eq 1){
            $conn = $_.OLEDBConnection.Connection
```

```
            $conn = $conn.ToString()
            if($conn -like "*SQL*"){
                  $conSql++
            }
      }
}

#Build an array with file name and number of connections.
$infoToExport = $book | select @{n="FileName";e={$_.FullName};},
                        @{n="NumberConnectionsSQL";e={$conSql};}
#Close the book.
$book.Close($false, [System.Type]::Missing, [System.Type]::Missing)

#Show information as a grid view.
$infoToExport | Out-GridView
```

As a result you should now see a grid with your result.

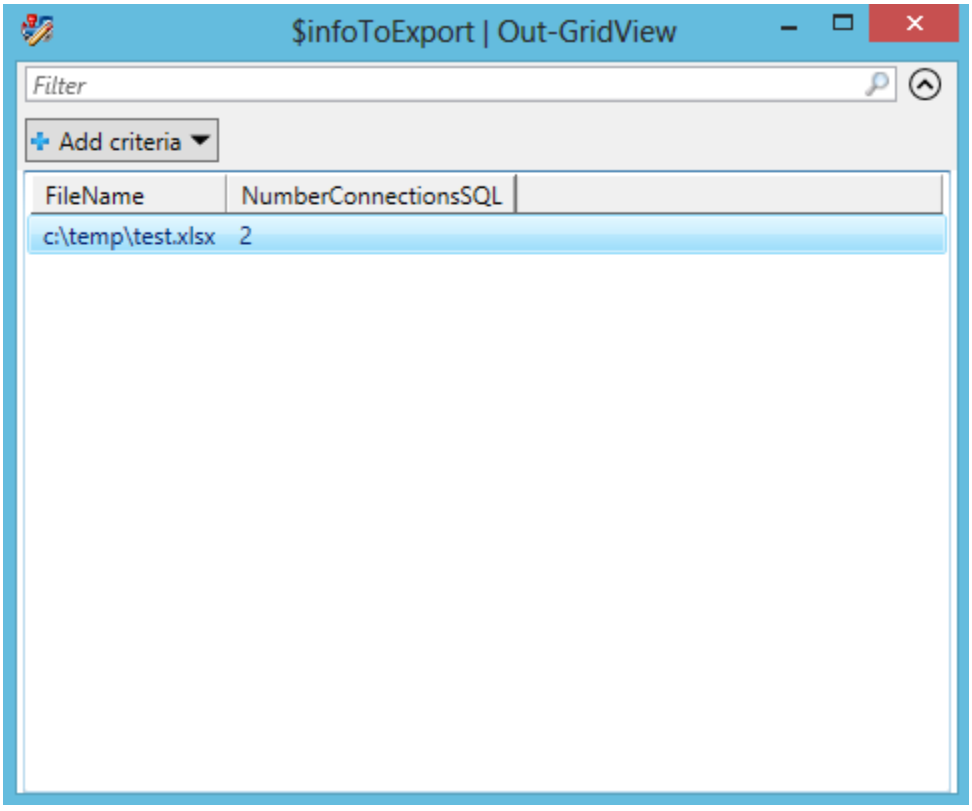


Figure 52: Result Grid

References

- Windows PowerShell Cookbook, Lee Holmes, 2010
- PowerShell PT
- MSDN: Parameter Attribute Declaration
- MSDN: Windows PowerShell