

PowerShell with SharePoint from Scratch

Exercises and Explanations



Peter and Kate Kalmström

PowerShell with SharePoint from Scratch

Peter and Kate Kalmström

Copyright © 2020 Peter Kalmström
All rights reserved

PowerShell with SharePoint from Scratch

Welcome to *PowerShell with SharePoint from Scratch*! This book gives the basic knowledge needed to start automating SharePoint business processes with PowerShell and to understand the code snippets available from Microsoft and other sources for administration of SharePoint farms and tenants.

PowerShell with SharePoint from Scratch is intended for SharePoint administrators, content creators and other power users who understand little or nothing about PowerShell coding but already know their way around SharePoint. (If you are new to SharePoint, study find my book *SharePoint Online from Scratch* first.)

I have been a developer in all my adult life, and when I started to write this book it became clear to me that I take too many things for granted when it comes to coding. I knew what I wanted to include in the book, but it was difficult for me to explain everything from scratch, which was what I wanted.

My aim was to write a good book on SharePoint management with PowerShell for people with little or no experience of writing code, but what was it that they did not know? What did I have to explain? I decided to ask my mother Kate for assistance.

Kate knows the SharePoint UI well, and she has worked a little with HTML, but she is definitely not a coder. She watched my video demonstrations and searched for information in books and online (that was often too advanced for her), but most of all she asked the right questions. "Why did you do that?", "What is this called?", "What is the difference between ...?"

I explained and explained, and piece by piece Kate put the text together. The result of our efforts is this book.

Kate is a former teacher and author of schoolbooks, so once she understands things, she also knows how to explain them in a pedagogic way. She has helped me with all my earlier books also, but this time she has taken a more active role in the text creation.

I have had the last word and approved of all the final text, and it is my "voice" you hear. And of course, I take the full responsibility for the technical

content and for the correctness of everything said about PowerShell in this book.

PowerShell with SharePoint from Scratch begins with a few chapters of basic information. After that, I give exercises with examples on how PowerShell scripts can be used to automate common SharePoint processes and how you can do things much quicker once you have created the script.

With each exercise, I introduce some new PowerShell theory, and I have tried to arrange the exercises in an order that gives a good learning progression.

PowerShell code can be written in many ways and still give the same results. This book builds on recommended practices and on my own experience after many years of managing SharePoint with PowerShell.

I used Windows 10, PowerShell 5.1 and SharePoint Online when I took the screenshots for this book, and when I recorded the video demonstrations I am referring to in most chapters.

However, I hope that *PowerShell with SharePoint from Scratch* will be useful even if you manage a SharePoint on-premises farm. Everything will not be the same in a farm, but I will point out some differences in the first chapters so you should still be able to understand the exercises

I have made it a priority to show code that is easy to read and understand. When you have gone through all the exercises in my book, I hope that you have enough knowledge to continue exploring the PowerShell possibilities on your own.

Good luck with your studies!

Peter Kalmström

Table of Contents

BASICS

1 WHY AUTOMATE?

1.1 ACCURACY

1.2 TRACKING

1.3 SPEED

1.4 METHODS

2 POWERSHELL INTRO

2.1 TECHNIQUE

2.2 CODING OPTIONS

2.3 POWERSHELL ISE

2.3.1 Enter Code

2.3.2 Future of PowerShell ISE

2.4 COMMANDS

2.4.1 Cmdlets

2.4.1.1 Parameters and Values

2.4.1.2 Command Panes

2.4.1.3 Get-Command

2.4.1.4 Operators

2.5 SCRIPTS

2.5.1 Save a Script

2.5.2 Run a Script

2.5.2.1 Allow Scripts to be Run

2.5.2.2 Run from PowerShell ISE

2.5.2.3 Run from Windows Explorer

2.6 HELP

2.6.1 Test Often!

2.6.2 Help in the Script Pane

2.6.3 The Help Dropdown

[2.6.4 Help in the Console pane](#)

[2.6.4.1 At Script Run](#)

[2.6.4.2 Help cmdlets](#)

[2.6.4.3 Get Info](#)

[2.6.4.4 Write-Host](#)

[2.7 SUMMARY](#)

[3 FIRST USE OF POWERSHELL WITH SHAREPOINT](#)

[3.1 SHAREPOINTPNPPOWERSHELL](#)

[3.1.1 Find SharePointPnPPowerShell](#)

[3.1.2 Install SharePointPnPPowerShell](#)

[3.1.3 Import SharePointPnPPowerShell](#)

[3.2 CONNECT TO SHAREPOINT](#)

[3.2.1 Check Connection](#)

[3.2.2 Comment and comment out](#)

[3.3 SUMMARY](#)

[EXERCISES](#)

[THEORY](#)

[STEPS](#)

[DEFINITIONS](#)

[MEANING OF PARAMETER/ARGUMENT AND VARIABLE](#)

[CMDLET DIFFERENCES IN SHAREPOINT EDITIONS](#)

[4 AVOID THE LOGIN](#)

[4.1 THEORY](#)

[4.1.1 String](#)

[4.1.2 Variables](#)

[4.1.2.1 Create and Declare a Variable](#)

[4.1.2.2 Assign the Variable a Value](#)

[4.1.2.3 Initialize a Variable](#)

[4.1.2.4 Assignment Operators](#)

[4.1.2.5 Properties](#)

[4.1.2.6 Ask with a Variable](#)

[4.1.2.7 See all Variables](#)

[4.2 STORE URL IN VARIABLE](#)

[4.2.1 Steps: Store URL in Variable](#)

[4.3 SAVE THE CREDENTIALS](#)

[4.3.1 Save the Browser Login](#)

[4.3.2 Save in Windows Credential Manager](#)

[4.3.2.1 Steps](#)

[4.4 SUMMARY](#)

[5 CREATE AND REMOVE A SHAREPOINT APP](#)

[5.1 THEORY](#)

[5.1.1 Apps = Lists](#)

[5.1.2 Command Pane and IntelliSense](#)

[5.1.3 Unnamed Parameter Values](#)

[5.2 CREATE APP](#)

[5.2.1 Steps](#)

[5.3 REMOVE APP](#)

[5.3.1 Steps](#)

[5.4 SUMMARY](#)

[6 FUNCTION FOR APP CREATION](#)

[6.1 THEORY](#)

[6.1.1 Functions](#)

[6.1.1.1 Create a Function – Name](#)

[6.1.1.2 Create a Function - Commands](#)

[6.1.1.3 Create a Function - Input Parameters](#)

[6.1.1.4 Initialize a Function](#)

[6.1.1.5 Call a Function](#)

[6.1.1.6 Expand and Collapse Functions](#)

[6.1.2 Write-Host](#)

[6.1.2.1 Addition Operator and Parenthesis](#)

[6.2 STEPS](#)

[6.2.1 Create a Function](#)

[6.2.2 Call the Function](#)

[6.2.3 Add Progress Message](#)

[6.3 SUMMARY](#)

7 CREATE MULTIPLE APPS WITH FOR-LOOP

[7.1 THEORY](#)

[7.1.1 Less Than and Increase Operators](#)

[7.1.2 For-Loop](#)

[7.2 LIST CREATION](#)

[7.2.1 Steps](#)

[7.3 LIST REMOVAL FOR-LOOP](#)

[7.3.1 Steps](#)

[7.4 SUMMARY](#)

8 CODE CREATION IN EXCEL - MULTIPLE CREATE-MYLIST CALLS

[8.1 PREREQUISITES](#)

[8.2 THEORY](#)

[8.3 STEPS](#)

[8.4 REMOVE APPS](#)

[8.5 SUMMARY](#)

9 CODE CREATION IN EXCEL - CONTENT STRING

[9.1 THEORY](#)

[9.1.1 Array](#)

[9.1.1.1 PowerShell Indexing](#)

[9.1.2 TEXTJOIN](#)

[9.1.3 Foreach Loop](#)

[9.2 STEPS](#)

[9.2.1 Create a String](#)

[9.2.2 Create an Array](#)

[9.2.3 Use the Array in a Foreach Loop](#)

[9.3 REMOVE APPS](#)

[9.4 SUMMARY](#)

[10 FUNCTION ENHANCEMENTS](#)

[10.1 THEORY](#)

[10.1.1 Type a Variable](#)

[10.1.2 If Statement](#)

[10.1.3 Methods](#)

[10.2 STEPS](#)

[10.2.1 Type the Function Variables](#)

[10.2.2 If Statement](#)

[10.2.3 Call the Function](#)

[10.3 SUMMARY](#)

[11 CREATE DOCUMENT LIBRARIES](#)

[11.1 THEORY](#)

[11.2 STEPS](#)

[11.3 SUMMARY](#)

[12 CREATE A GENERAL APP FUNCTION](#)

[12.1 FUNCTION THAT CREATES ANY APP](#)

[12.1.1 Theory](#)

[12.1.1.1 Specify the App Type with a General Type](#)

[12.1.1.2 Convert the AppType Parameter to a String](#)

[12.1.1.3 Call a Function for All App Types](#)

[12.1.2 Steps](#)

[12.2 CALL FROM FUNCTION TO FUNCTION](#)

[12.2.1 Steps](#)

[12.3 SUMMARY](#)

[13 CREATE AND REMOVE SITE COLLECTIONS](#)

[13.1 CREATE SITE COLLECTION WITH](#)

COMMUNICATION SITE

13.1.1 Theory

13.1.1.1 Select Cmdlet from Command Pane

13.1.1.2 Get-Help

13.1.2 Steps

13.2 REMOVE SITES

13.2.1 Steps

13.3 SUMMARY

14 IMPORT DATA FROM EXCEL - ONE COLUMN

14.1 THEORY

14.2 STEPS

14.3 SUMMARY

15 IMPORT DATA FROM EXCEL - MULTIPLE COLUMNS

15.1 THEORY

15.1.1 The += Operator and Key-Value Pairs

15.1.2 Get-Content and Text Files

15.1.3 ConvertFrom-Csv

15.1.4 Get-PnPFields

15.2 STEPS

15.3 SUMMARY

16 ADD FILES TO SHAREPOINT – UPLOAD FILES

16.1 THEORY

16.1.1 Get-ChildItem

16.1.2 Add-PnPFile

16.1.3 Foreach Loops and IntelliSense

16.2 STEPS

16.3 SUMMARY

17 ADD FILES TO SHAREPOINT – MOVE UPLOADED FILES

17.1 THEORY

17.1.1 Error handling

17.1.2 Find Folder

17.1.3 Comparison Operators

17.2 STEPS

17.3 SUMMARY

18 AUTOMATE A POWERSHELL TASK

18.1 THEORY

18.2 STEPS

18.3 SUMMARY

19 UPLOAD FILES WITH METADATA – LIBRARY

19.1 PREREQUISITES

19.2 THEORY

19.3 STEPS

19.4 SUMMARY

20 UPLOAD FILES WITH METADATA – STRUCTURE

20.1 THEORY

20.1.1 Split Operator

20.1.2 Class

20.1.2.1 Create a Class

20.1.2.2 Create an Instance of a Class

20.2 STEPS

20.2.1 Create a CV Class

20.2.2 Create an AllCVFiles Array

20.2.3 Create a Foreach Loop for the Array

20.2.4 Get MetaData for the Decision Column

20.2.5 Split out the Department from the File Name

20.2.6 Split the First and Last Name in the File Name

20.3 SUMMARY

21 UPLOAD FILES WITH METADATA – UPLOAD

21.1 THEORY

21.2 STEPS

21.3 SUMMARY

22 CREATE A SHAREPOINT INTRANET - SUBSITES

22.1 PREREQUISITES

22.2 CREATE SUBSITES

22.2.1 Theory

22.2.2 Steps

22.3 REMOVE SUBSITES

22.3.1 Theory

22.3.2 Steps

22.4 SUMMARY

23 CREATE A SHAREPOINT INTRANET - NAVIGATION

23.1 THEORY

23.2 STEPS

23.2.1 Subsite links on the HQ site

23.2.2 Links on the subsites

23.2.3 Link to an external site on the HQ site

23.3 REMOVE EXTERNAL LINK

23.3.1 Theory

23.3.2 Steps

23.4 SUMMARY

24 CREATE A SHAREPOINT INTRANET – APPS

24.1 THEORY

24.2 STEPS

24.3 SUMMARY

25 CREATE A SHAREPOINT INTRANET – FUNCTIONS

25.1 THEORY

25.2 ADD REMOVAL CODE TO FUNCTION

25.2.1 Steps

25.3 ADD APP AND NAVIGATION CODE TO FUNCTION

25.3.1 Steps

25.4 SUMMARY

26 CREATE A SHAREPOINT INTRANET – PAGES

26.1 THEORY

26.1.1 Modern Page

26.1.2 Classic Page

26.2 STEPS

26.3 SUMMARY

27 CREATE A SHAREPOINT INTRANET – PROGRESS MESSAGES

27.1 THEORY

27.2 STEPS

27.3 SUMMARY

28 CREATE A SHAREPOINT INTRANET – THEMES

28.1 THEORY

28.2 STEPS

28.2.1 Function for Tenant Themes

28.2.2 Add Themes to Subsites

28.2.3 Remove Themes

28.3 SUMMARY

29 CREATE A SHAREPOINT INTRANET – ADD VIDEO

29.1 THEORY

29.2 STEPS

29.3 SUMMARY

TO CONTINUE

SCRIPTS

ABOUT THE AUTHORS

BASICS

You can find a lot of information about PowerShell online, but it is often written for people who are already experienced coders. There is very little for people who are not used to coding but eager to learn, and that is the major reason why I wrote this book.

I will assume that you are an experienced computer user and SharePoint manager, but I will *not* assume that you already know some PowerShell coding or have worked with another scripting language. I will explain all details as well as I can, so that you will develop a good foundation for work with PowerShell by studying this book.

In the first part of the book, I will explain a few fundamental facts that you should understand or at least know where to find in this book so that you can go back if needed. As soon as possible we will go over to practical code writing, but before that you need to know:

- Why we should automate business processes in the first place.
- What PowerShell is and which tool we will use to work with it.
- What a script is, how it is executed and saved and what the most important parts are called.
- How you can get help inside the PowerShell tool.
- How to install the module that is recommended when you want to use PowerShell with SharePoint.

That is all! The rest of the basic PowerShell theory you will learn in this book will be introduced in the context of an exercise.

1 WHY AUTOMATE?

Before we start looking into the scripts, I will point out why you should automate business processes in the first place. As you are reading this book, you probably already know why, but maybe you will still find it useful to have my opinion on the benefits of SharePoint automation compared to manual processes.

You might also find these points relevant in discussions with others. "Why don't we just do this manually?" is still a common question.

1.1 ACCURACY

Most organizations have processes that need to be performed in a specific way and order, and the best way to make sure that they are really performed this way, is to automate them. When processes are performed manually, you can never be sure that everything is done 100% correctly.

People make mistakes. Even if you check and double-check your work there will be errors, and these are often difficult to find. People also get bored and are prone to do variations, which might lead to further mistakes.

A PowerShell script, on the other hand, always works in the same way. Once you have tested the script and made sure that it works as it should, you can rely on its accuracy. The script follows the same process. in the same way, every time. It does not make mistakes or variations now and then, like we humans do.

Another factor is that human errors often are random, which makes them hard to find and correct. If a PowerShell script does something wrong, it at least does it consistently wrong – making the same mistake in every run.

1.2 TRACKING

With a PowerShell script, it is easy to track processes. It can log and document what has been done, something that is often requested by the management and sometimes even by law. Such tasks are often tedious and boring to perform manually and tend to be performed insufficiently or not at

all when done manually.

When you let PowerShell scripts keep track of what happens, you will also have documentation. A script can for example add items in a SharePoint list or comments in a list column describing what has been done and why.

1.3 SPEED

It takes time to build a PowerShell script, but the script creation time will be well spent if you have a process that must be performed repeatedly. Once the script has been tested and works well, no more time must be spent on the process, but to do it manually takes time over and over again.

Furthermore, the PowerShell script can perform a task much quicker than a human can do.

1.4 METHODS

There are multiple ways to automate SharePoint, with and without writing code. In my books *SharePoint Flows from Scratch* and *SharePoint Workflows from Scratch*, I introduce two of the most common and popular no-code methods.

In my work as a consultant I often prefer coded solutions like PowerShell, as they give more control and are easier to support. To have PowerShell scripts running on a separate server or in an Azure function is very powerful.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/Why-Automate.htm>

2 POWERSHELL INTRO

Windows PowerShell is a Windows command-line shell that many system administrators find useful for different kinds of automation. PowerShell has numerous building blocks, and it is up to you to combine them into something that accomplishes a given task.

In this book, we will focus on how SharePoint can be managed with PowerShell, so that the code we create in PowerShell will make something happen in SharePoint.

In PowerShell, we are working with a developer interface (API) and not in a graphic user interface (GUI) like the one users see in SharePoint. This requires a different way to work, but it also means that some terms are used differently and that things are named in different ways. I will inform about those differences when we meet them in a relevant context.

I assume that you want to get started with the programming, so I will not burden you with a lot of theory in the beginning of this book. Instead, each of the exercises that consist the major part of the book will have theory sections where I explain theory that is practiced in that exercise.

Here I will just point out a few basics that you need to understand about the PowerShell scripting framework before we can continue with the coding, such as what app to use and how to save and run PowerShell scripts.

I will also explain a few basic PowerShell concepts, but the rest of the basics will be introduced in the context of exercises.

I will however start with a short section with technical details on PowerShell. They are not strictly necessary for the understanding of this book, so if you don't find them interesting, you can go directly to section 2.2.

2.1 TECHNIQUE

When you talk about a shell in connection with computing, you refer to a user interface that gives access to the services of an operating system. In general, operating system shells use either a command-line interface or a graphical user interface, and it is named a shell because it is the outermost layer around the operating system kernel.

PowerShell includes an interactive prompt and a scripting environment. These two can be either combined or used independently.

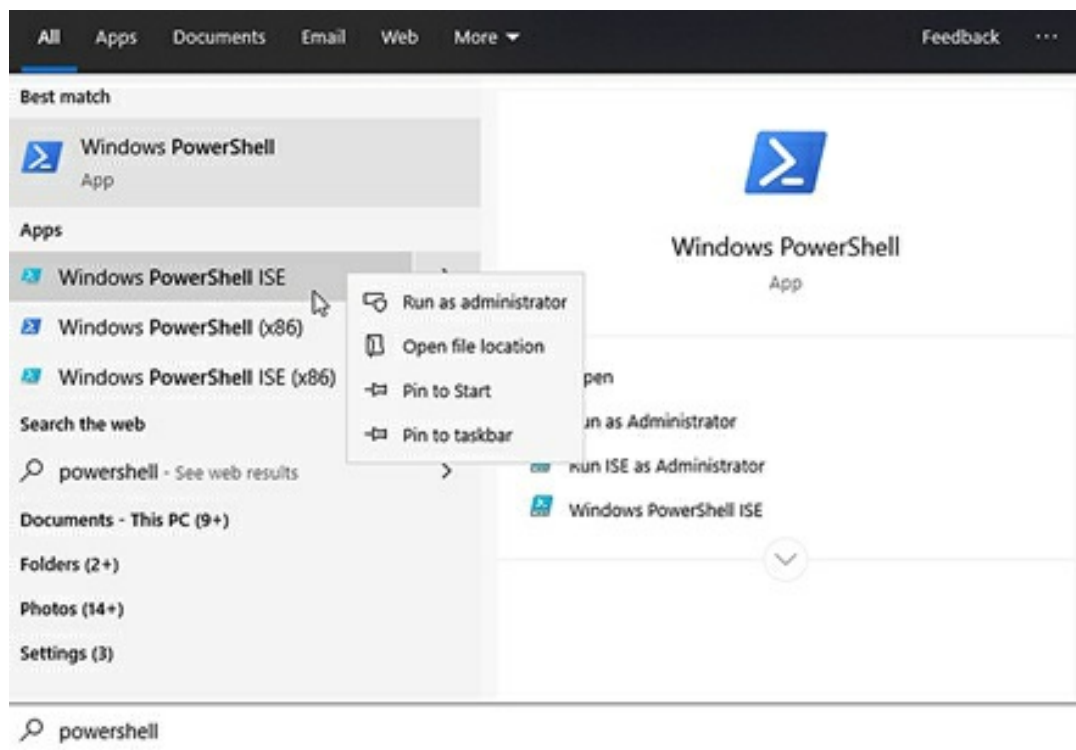
Most shells accept and return text, but PowerShell is built on top of the .NET Framework common language runtime (CLR) and the .NET Framework and accepts and returns .NET Framework objects. This means that you can configure and manage Windows and automate SharePoint with entirely new tools and methods.

Like many shells, PowerShell gives you access to the computer's file system, but PowerShell also gives easy access other data stores, such as the registry and the digital signature certificate store.

PowerShell provides full access to [COM](#) and [WMI](#), so that administrators can perform administrative tasks on both local and remote Windows systems.

2.2 CODING OPTIONS

Windows offers both a command-line option (Windows PowerShell), and an Integrated Scripting Environment (PowerShell ISE). You will find both when you click on the Windows icon in Windows 10 and start typing PowerShell.



In earlier versions, only Windows PowerShell is by default visible under Apps, while PowerShell ISE is hidden in the Administrative tools. You can find it in the Control Panel, by searching for PowerShell ISE on the Start screen or typing it at the command prompt.

PowerShell scripts can be created and executed in Windows PowerShell, but the PowerShell ISE makes it easier for you to work with script files. If you just want to execute one or a few commands, Windows PowerShell will work fine, but for more extensive work PowerShell ISE makes everything easier.

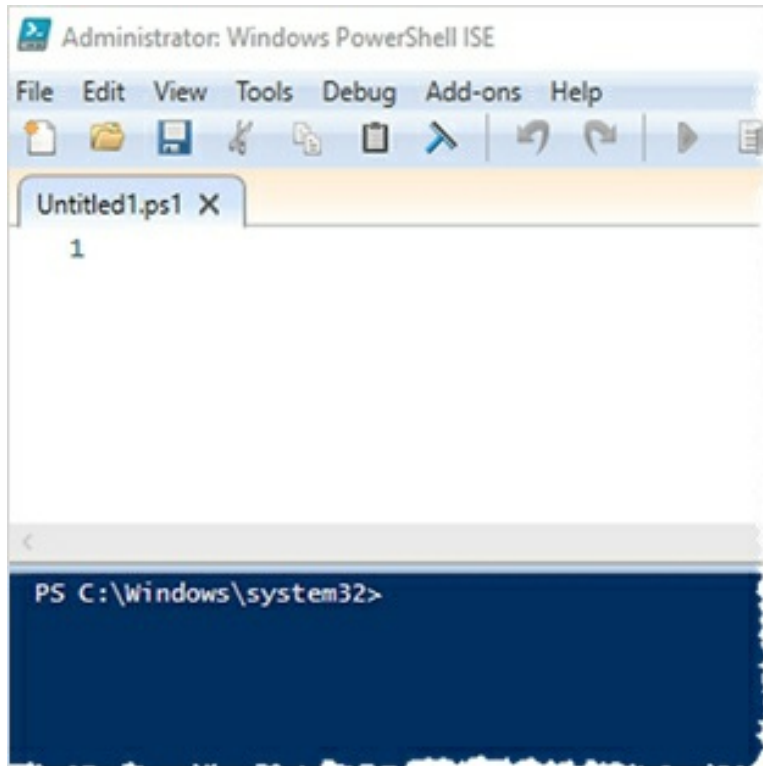
In this book, we will work in PowerShell ISE. With PowerShell ISE, we will have color coding, IntelliSense and other features that makes scripting easier and less prone to errors.

2.3 POWERSHELL ISE

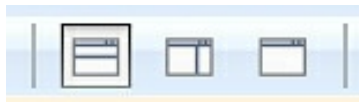
The first time you use Windows PowerShell ISE, you must run the app as an administrator. This is important, because you need to manipulate your Windows environment.

PowerShell ISE has two main parts:

- The script pane, also called file pane, where the scripts are composed and can be saved to a file.
- The console pane, or output pane, with the command prompt.

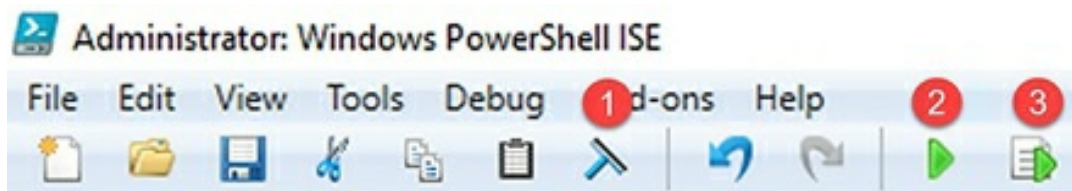


By default, the script pane is placed above the console pane, but you can change that with buttons in the toolbar.



The script is entered to the console pane when you execute it, together with any error messages and other information. I will explain later how you can influence what is shown in the console pane when code is executed.

PowerShell ISE has a toolbar on top. The icons below the menu bar has tooltips, so I will just mention three of them here.

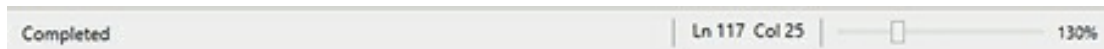


1. Clear the console pane. This is especially useful when you are learning PowerShell, because the console pane often gets messy when you try different things that might not all turn out well.

You can also clear the console pane by entering `cls` at the command prompt.

2. Execute the whole script.
3. Execute a part of the script.

Below the toolbar comes the script and console panes, and at the bottom, you can find a status bar and a text size slider.



Overall, Microsoft is helpful with hints on keyboard shortcuts in dropdowns and tooltips, so with a few exceptions I will not mention them in this book.

You will see the hints when you work with PowerShell ISE, and then it is up to you if you want to use the shortcuts instead. If you will be using PowerShell extensively, I recommend that you learn them!

2.3.1 Enter Code

PowerShell ISE has some functions that help you get the code correct:

- The code is automatically color coded to be easier to follow.
- Code with errors that PowerShell can discover are marked with a red line.
- Suggestions for autocompletion are given in a systematic way that I will explain later.

PowerShell code can be entered at the command prompt in the console pane, and that method is often used for commands that should not be kept in the script. You can also use the command prompt to get information about variables, *refer to chapter 4*.

More complex commands are normally typed into the script pane, and you must use the script pane for everything that should be saved. The code in the console pane will not be there when you open the file next time, even if you have saved the script.

When you enter code – in any programming language – you should always consider the readability, both for your own sake and for other people who might be using the code. You should also make it a habit to add comments to

the code, see next chapter.

2.3.2 Future of PowerShell ISE

PowerShell ISE is supported in all versions of Windows PowerShell up to and including Windows PowerShell V5.1, but the app is in maintenance mode and no new features are likely to be added.

Version 6 of PowerShell is cross-platform (Windows, MacOS and Linux), open-source and built for heterogeneous environments and the hybrid cloud.

PowerShell V6 does not support ISE, so if you need that version and want a graphical tool for PowerShell scripts, I recommend that you use Visual Studio Code.

Even if I myself use Visual Studio for the complex PowerShell scripts I write in my work as a developer and consultant, I have decided to use the ISE app rather than Visual Studio for this book.

When we use PowerShell ISE we can focus on the code, and readers don't have to learn a scripting tool that is considerably more complicated.

Moreover, the current PowerShell version, 5.1, that is included in Windows 10, will probably live alongside version 6 for a long time to come.

Should you prefer going over to Visual Studio later, Microsoft has given recommendations on how to simplify the user interface of Visual Studio Code to look more like PowerShell ISE: <https://docs.microsoft.com/en-us/powershell/scripting/components/vscode/how-to-replicate-the-ise-experience-in-vscode?view=powershell-6>.

2.4 COMMANDS

A command is a piece of code that tells PowerShell what to do. Commands often have several building blocks, and in this book, you will learn how to create such commands. The first command will actually come already in this chapter, in 2.5.3.1.

`Set-ExecutionPolicy -ExecutionPolicy Unrestricted`

The command above is built from three blocks: a cmdlet, a parameter name and a parameter value. You will learn more about them below.

I recommend that you write each command on a new row. Press Enter on the keyboard to add a new row in PowerShell ISE.

There are certain standards on how to write PowerShell code when it comes to upper and lower case, and I follow it below, but in general the code is not case-sensitive.

When you write a script, you must remember that the PowerShell commands are executed in row order, the order they are placed in the script. This means that I, to get a good progression for my explanations, sometimes will tell you to enter new code above already existing code.

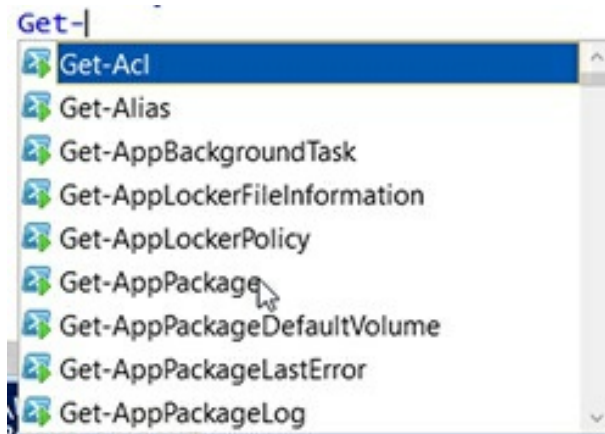
2.4.1 Cmdlets

A PowerShell command always begins with a cmdlet (pronounced "command-let"). Some cmdlets can be used alone as full commands, but cmdlets become really powerful when you combine them with other code blocks into commands that automate complex tasks.

A cmdlet name is built of two parts. The first part is most often a verb that tells what to do, and the second part is a noun that tells what to do it with. 'Get-Content' is such a cmdlet. Get is the verb, Content is the noun.

In some cases, a preposition is added directly to the verb, as in ConvertTo-Csv, and the adjective New can be used instead of the verb, as in New-PnPList. The noun part can consist of several nouns that are written in CamelCase pattern: Set-ExecutionPolicy.

The two parts of the cmdlet are always separated with a hyphen. PowerShell ISE will give suggestions for completion of the cmdlet when you have typed the first part and the hyphen.



You can also get cmdlet suggestions if you enter the first part of the cmdlet and then press Ctrl + Space on the keyboard.

PowerShell includes many basic core cmdlets, and they most often have other, hidden, cmdlets encapsulated. For example, the cmdlet Import-Module, has hidden cmdlets that tell PowerShell what actions to perform to import the module.

You can create your own cmdlets, and I recommend that you do it. Then you must specify each step that should be performed – you must create a function. This is described in chapter 6.

In this book, when I mention a cmdlet as *the Add-Content cmdlet* I am only referring to the cmdlet itself.

When I say *the Add-Content command*, I refer to the PowerShell command that starts with the Add-Content cmdlet.

2.4.1.1 Parameters and Values

The cmdlet is often followed by a space and a parameter that specifies how the cmdlet should work and what it should do. Another word for parameter is argument, but I will use parameter in this book.

Parameters often – but not always – have values, and in some cases only the value and not the parameter name is specified after the cmdlet. (I will explain more about that later.)

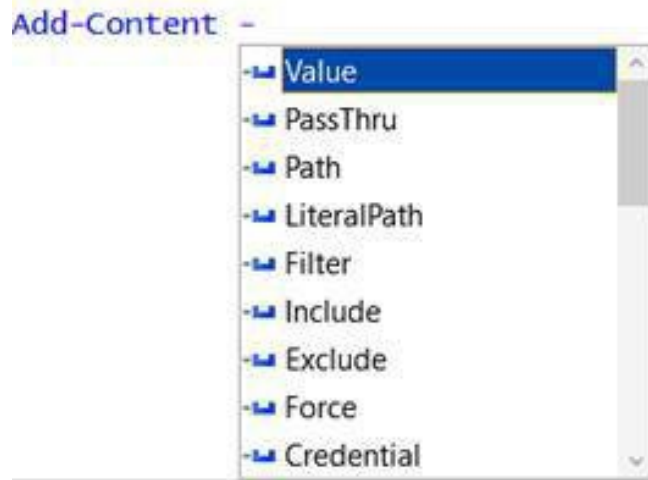
Sometimes parameters are mandatory, and in these cases the cmdlet cannot be used if you have not specified the values of all mandatory parameters.

When parameters are written out, they always have a hyphen before the

name: -Type.

The value of the parameter comes after the parameter name and a space, and it can be a hardcoded value or a variable. The value can also be specified by another cmdlet.

When you enter a space and a hyphen after a cmdlet, or after another parameter in a command, parameter names for that cmdlet will be suggested.

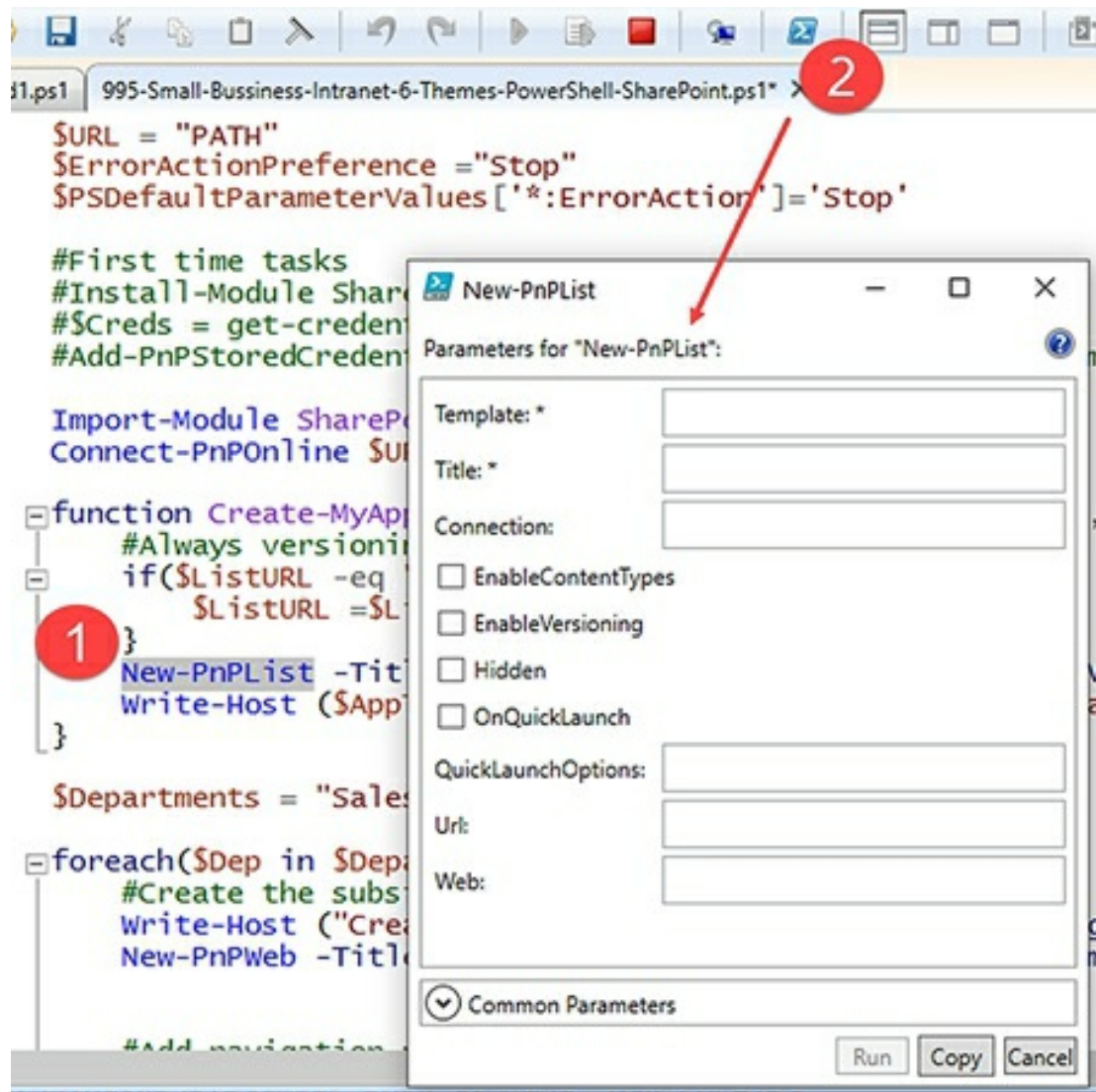


When you add a space after the parameter, you will sometimes have value suggestions. Other PowerShell building blocks can also have IntelliSense suggestions.

2.4.1.2 Command Panes

There are lists of cmdlets online, but the best help to find the correct cmdlet is given inside PowerShell ISE. On the right side, you can open a Command Add-on, where you can search for a suitable cmdlet and read information on how each cmdlet is used.

When you select – or put the cursor inside – a cmdlet in the script pane or console pane and then open the Command Window, information about that specific cmdlet is displayed in the Command Window.



When no cmdlet is selected, the Command Window works in the same way as the Command Add-on. Therefore, and I will call both "command pane" in this book.

If you know which cmdlet to use, the Command Window is the quickest way to get information about it. If you are searching for the suitable cmdlet to use, both these command panes are helpful.

You can open the command panes with buttons in the toolbar.

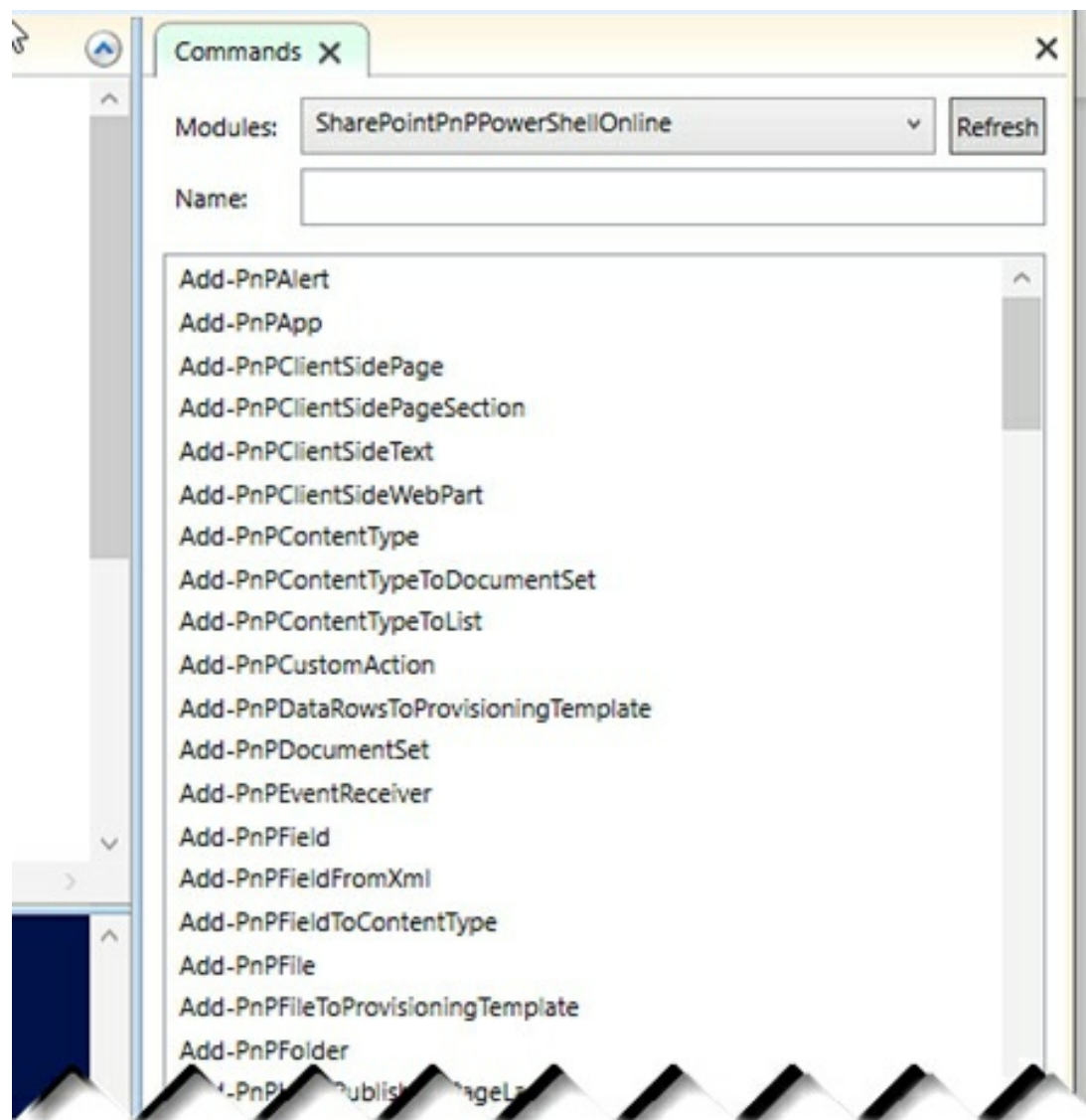


The Command Add-on can also be opened from the View menu, and it is displayed to the right of the two other panes in PowerShell ISE.

The Command Window can also be opened from the Help menu or by running the cmdlet Show-Command, and it is displayed as a dialog window.

By default, all cmdlets are shown in the Command Add-on and in the Command Window when no cmdlet is selected. When you start writing, the suggestions will become fewer the more you write. You can very well search for a word, for example site, and get all cmdlets that handles sites in some way.

The cmdlets can also be filtered by module, and when you work with SharePoint, you often want to select the SharePointPnP PowerShell module before you start searching for a cmdlet.

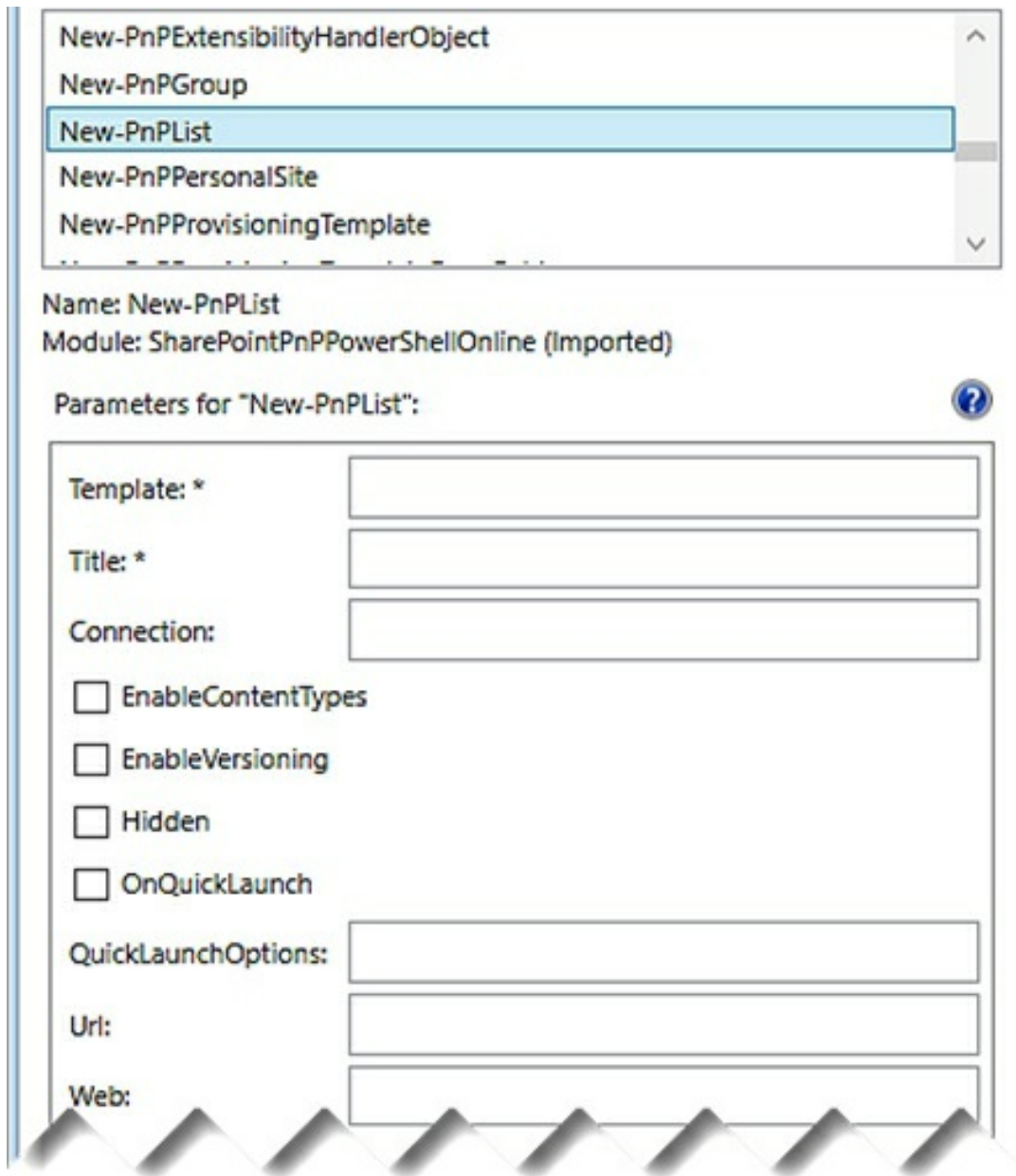


Select one of the cmdlets in the command pane to see common parameters for the cmdlet. Mandatory parameters are marked with a star, and you must add values for them when you use this cmdlet.

I recommend that you use the command panes to find the right cmdlet and see parameters for it. However, when you enter the code you should use the script pane. I will come back to that in chapter 5.

Click on the information icon to see examples on how the cmdlet and the parameters are used.

When you hover over a parameter name, you will also have some information on how it is used.



As you see in the image above, the command pane has check boxes for common parameters. However, when you check a box and click on Insert, your parameters will be added in the console pane, not to the script pane where I have recommended that you should work.

Another way to get more information about how a cmdlet is used, is the Get-Help cmdlet, *refer to* chapter 13.

2.4.1.3 Get-Command

Should you not find the cmdlet you are looking for among the PowerShell ISE suggestions, you can type `Get-Command` at the command prompt and press Enter. You will then get a complete list of all cmdlets that PowerShell understands. If you import a new module (which contains new cmdlets) that list will expand.

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

2.4.1.4 Operators

PowerShell uses many different so-called operators. These are characters in the code that PowerShell interprets in a specific way. We will use several such operators in this book. An example is the addition operator. It is written with a plus sign, and PowerShell interprets it to signify exactly that – an addition.

Some operators consist of just a single character, like `+`, while others are multiple characters as `-eq`, `-or` and `-not`.

Here you can also see that some operators start with a hyphen, just like parameters. However, parameters often have values, but operators can never have values connected to them.

It depends on the context if PowerShell interprets one or several characters as operators. If the plus is written within quotation marks, for example, it will not be considered an operator. Instead, PowerShell will treat the plus as a string, *see* 4.1.1.

PowerShell operators fall into several categories, and you will learn some of them when you study the exercises in this book.

2.5 SCRIPTS

When you combine several PowerShell commands and other PowerShell building blocks, you have created a script that can be saved as a text file with the `.ps1` extension. Scripts can be executed to perform one or multiple actions.

Sometimes a block of PowerShell code is also called a script, even if it is not

saved in a .ps1 file.

2.5.1 Save a Script

PowerShell scripts in PowerShell ISE are saved in the same way as Office files. Therefore, to save a script you can either press CTRL+S, click on the Save icon in the toolbar or select 'File' and 'Save'. If it is the first time you save the script, you will be asked to give the script a name.

You can of course also Save a script As, and that is very useful when you want to reuse code. Open an existing script, change it for the new purpose and save it with another name.

An asterisk after the script name on a tab in PowerShell ISE shows that the file has not been saved since it was changed. The asterisk disappears when the file is saved.

When you have saved and closed a script and then open it again, the console pane will be clean. Only the script area will look as it did when you saved it.

2.5.2 Run a Script

When you run a PowerShell script (or part of a script), the actions defined in the code you run will be executed in the order they are written.

To avoid accidents, .ps1 files cannot be run by double-click. Instead they are executed from within PowerShell.

When you have run a script, or a part of it, and want to check the result in SharePoint, remember that you must refresh the page to see the change.

2.5.2.1 Allow Scripts to be Run

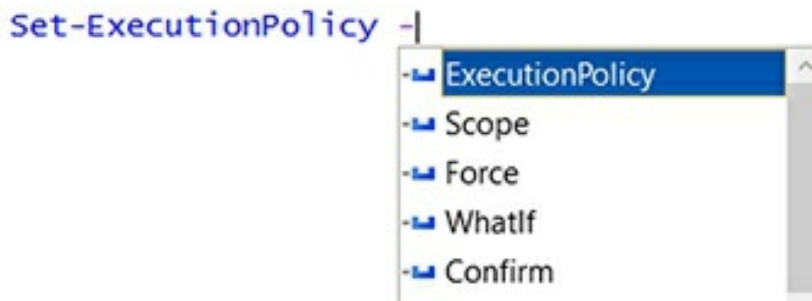
Running script files on a Windows computer is not allowed by default, so before you can run any scripts you must allow that by changing the execution policy. The cmdlet for that is Set-ExecutionPolicy. I usually set my policy to 'Unrestricted'.

Follow these steps to set the execution policy:

1. Enter Set-ExecutionPolicy in the PowerShell ISE script pane. When you have typed Set- you will have IntelliSense and can choose

ExecutionPolicy.

2. Add a space and a hyphen to get suggestions for the parameter.
3. Select ExecutionPolicy.



4. Add a space, and you will have suggestions for the value of the ExecutionPolicy parameter.
5. If you have selected the value Unrestricted, the command will look like this:

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

6. Run the script row, see below.

2.5.2.2 Run from PowerShell ISE

Scripts can of course be run directly from within PowerShell ISE, without being saved, and that is what you do when you create and test scripts.

PowerShell ISE even allows you to run only a part of the script, which we will do a lot for testing blocks of code in the exercises in this book.

In PowerShell ISE, you can run PowerShell code in these ways:

- Run the row in the script pane where you have placed the cursor, by clicking on the Run Selection icon in the toolbar or by pressing the F8 key.
- Run a selected part of the script in the script pane by clicking on the Run Selection icon in the toolbar or by pressing the F8 key.
- Run all code in the script pane by clicking on the Run icon in the toolbar or pressing the F5 key.
- When code is added at the command prompt, press Enter on the keyboard to

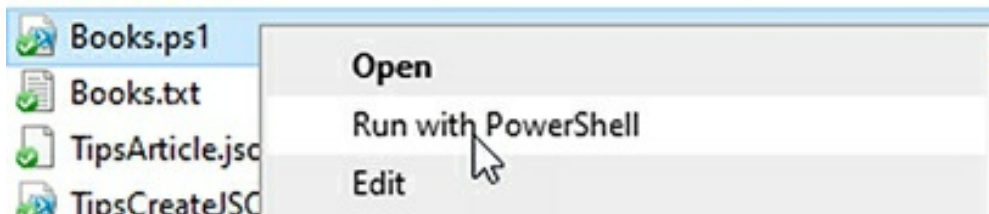
run it.

You can see that the script is running at the bottom of the screen. For a longer script, you can also see how the console pane fills up with text. If there is an error, it will be displayed in the console pane with an explanation.



2.5.2.3 Run from Windows Explorer

If you want to run a script file without opening it, right-click on the file and select 'Run with PowerShell'.



(If you select 'Open', the file will open in Notepad, and if you select 'Edit' it will open in PowerShell ISE.)

2.6 HELP

The information you get in the various tips I give below is not always easy to follow for a new beginner. I of course hope that my instructions will be so clear that you don't make any mistakes, but at the same time I am sure that you will make them! I still make a lot of mistakes myself, believe me, but after many years of writing code I know how to quickly find the errors.

I have already introduced the command panes, in section 2.4.1.2, where you can get a lot of information. In this section, I will give you some more tips on how you can find information and avoid extensive troubleshooting.

My reason for having this section so early in the book is that the information is important, and I want to make sure that you know where to find it.

2.6.1 Test Often!

Let me first share my best tip for avoiding extensive troubleshooting of code: test as often as possible. If you test each little piece of script before you continue adding new commands, you will discover errors and will immediately understand where to look for and correct them.

Testing often makes it easier to troubleshoot, and errors are discovered early in the script creation and will hopefully not be repeated.

That is how I do in this book and in the video demonstrations I refer to. Any errors will be quickly pointed out in the PowerShell ISE console pane, and it is easier to understand the explanation there if I don't need to go through a long script for the rows that contain errors.

Testing often requires a way to easily get rid of things that my script has created, so I always write removal code. You will find several examples on these in the exercises.

2.6.2 Help in the Script Pane

You can avoid many errors by working in the PowerShell ISE script pane. When this tool discovers errors in the code, it marks them with red underlining, and when you hover over the red mark you will have a hint about the error.

```
foreach($Dep in $Departments){  
    #Create the subsite  
    New-PnPWeb -Title $Dep -u
```



I have already mentioned that the PowerShell ISE script pane has IntelliSense, and this is very useful. It lets you see what is possible to add to different objects, and by selecting instead of typing you will avoid typos. I will explain how to get the IntelliSense throughout the exercises in this book.

2.6.3 The Help Dropdown

The PowerShell ISE menu bar has a Help dropdown with three options.



The first option opens a Microsoft website with a lot of information about PowerShell.

The second option lets you run the Update-Help cmdlet, *refer to Get-Help* below.

The third option opens the Command Window, *refer to Command Panes*.

2.6.4 Help in the Console pane

The PowerShell ISE console pane and command prompt are useful for information searching and troubleshooting.

2.6.4.1 At Script Run

When you run a script or a block of code, error information is always displayed in the console pane if the execution cannot be completed. This information is not always easy to understand, but at least you will learn exactly where in your code the error occurred.

```
Remove-PnPWeb : Object reference not set to an instance of an object.
At line:3 char:9
+ ~~~~~
+ ~~~~~
+ CategoryInfo          : WriteError: (:) [Remove-PnPWeb], NullReferenceException
+ FullyQualifiedErrorId : EXCEPTION,SharePointPnP.PowerShell.Commands.RemoveWeb
```

In the error message above, the error occurred on line 3, character 9, and the error has something to do with an object. The first thing to do, is of course to check what exactly is written on line 3, character 9. Is there an object being referenced there?

Troubleshooting is most likely where you will spend most of your time as a developer. Being stubborn and persistent are essential traits of a developer. If at first you don't succeed, try again and again and again. Of course, trying the same thing over and over again is the definition of madness, so make sure you try different things in a structured manner.

2.6.4.2 Help cmdlets

PowerPoint has a special help cmdlet, Get-Help. Enter it at the command prompt and add what you want to have help with, in the form of another cmdlet or just a word, for example list. Then information will be given in the console pane. I will explain more about the Get-Help cmdlet in chapter 13.

```
PS C:\windows\system32> Get-Help New-PnPList -Examples|
```

2.6.4.3 Get Info

When you type a variable at the command prompt and run it, you will have information about that variable, *refer to* chapter 4. This can help you find errors in the variable.

2.6.4.4 Write-Host

Use the Write-Host cmdlet in your code, to let PowerShell show a customized message when a block of code is executed. These messages make it easier to see where something has gone wrong. I will introduce this cmdlet in chapter 6 and then we will use it in many of the exercises.

```
PS C:\Users\PeterKalmström> Create-MyList -ListName "Hello World 3" -ListURL "HelloWorld3"  
List Hello World 3 has been created
```

2.7 SUMMARY

This chapter has given you a short overview of PowerShell, but I have tried to only explain what you need to know to understand and be able to work with next chapter.

I have also given some tips on how to get help. You will probably have reason to come back to them later, and you will surely need the help when you start creating your own scripts.

In the next chapter, I will describe how to start using PowerShell with SharePoint.

3 FIRST USE OF POWERSHELL WITH SHAREPOINT

The first time you use PowerShell with SharePoint, you should install a SharePoint module and create a script that connects to a SharePoint site. That is what we will do in this chapter.

3.1 SHAREPOINTPNPPOWERSHELL

When you use PowerShell with SharePoint, you need the SharePointPnPPowerShell module, a library of PowerShell cmdlets that allow you perform complex management actions towards SharePoint.

It is possible to work with SharePoint in PowerShell without the SharePointPnPPowerShell module, but it is not recommended. If you want to follow my instructions you must install the tool, because we will be using it throughout this book. It gives many benefits.

The SharePointPnPPowerShell module is installed on the computer and then imported to PowerShell each time it should be used.

3.1.1 Find SharePointPnPPowerShell

You can find the SharePointPnPPowerShell module at the PowerShell gallery, <https://www.powershellgallery.com>. Search for SharePointPnPPowerShell and filter by Module if necessary. Then click on the applicable edition.

The screenshot shows the PowerShell Gallery search results for 'SharePointPnPPowerShell'. The search bar at the top contains the text 'SharePointPnPPowerShell'. Below the search bar, there are filter options on the left and search results on the right. The search results list four modules: 'SharePointPnPPowerShellOnline', 'SharePointPnPPowerShell2016', 'SharePointPnPPowerShell2013', and 'SharePointPnPPowerShell2019'. Each result includes the number of downloads, the last updated date, and the latest version number.

If you have Windows 10, you can also get the SharePointPnPPowerShell module at <https://github.com/SharePoint/PnP-PowerShell> and at <https://docs.microsoft.com/en-us/powershell/sharepoint/sharepoint-pnp/sharepoint-pnp-cmdlets?view=sharepoint-ps>.

On both these sites, you will find download links for the SharePointPnPPowerShell that is suitable for your SharePoint edition.

SharePoint Version	Command to install
SharePoint Online	<code>Install-Module SharePointPnPPowerShellOnline</code>
SharePoint 2019	<code>Install-Module SharePointPnPPowerShell2019</code>
SharePoint 2016	<code>Install-Module SharePointPnPPowerShell2016</code>
SharePoint 2013	<code>Install-Module SharePointPnPPowerShell2013</code>

3.1.2 Install SharePointPnPPowerShell

The installation of the SharePointPnPPowerShell module is made directly in PowerShell, so when you have copied the "Install-Module..." text from the web page, paste it into the default Untitled.ps1 file in PowerShell ISE, that

you have opened as an administrator.

As I am using SharePoint Online for this book, the command will be:

```
Install-Module -Name SharePointPnPPowerShellOnline
```

For SharePoint 2016, the command looks like this instead:

```
Install-Module -Name SharePointPnPPowerShell2016
```

If you have not already changed your computer's execution policy, *refer to* Allow Scripts to be Run above, you must do that now. Otherwise you cannot perform the installation.

Run the installation command you pasted into PowerShell ISE, and the installation will start. It will take 1-2 minutes, and you can see at the bottom of the screen when it has been completed.

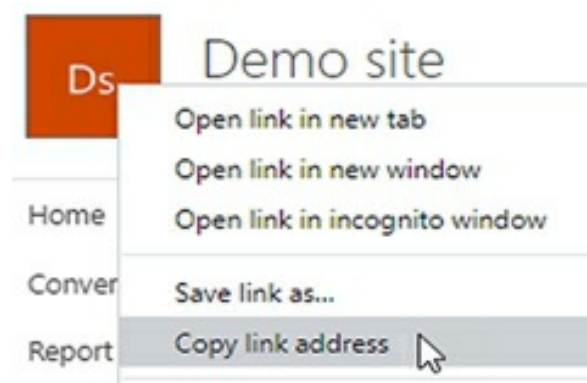
3.1.3 Import SharePointPnPPowerShell

When the SharePointPnPPowerShell module has been installed on the computer, you must import it each time you want to use it. This is done with the Import-Module cmdlet and the Name parameter. The value of the parameter should be your version of the SharePointPnPPowerShell module.

```
Import-Module -Name SharePointPnPPowerShellOnline
```

3.2 CONNECT TO SHAREPOINT

Now it is time to connect PowerShell to SharePoint. To get a clean link to the SharePoint site where you want to use your script, right-click on the site icon and select 'Copy link address'.



The cmdlet for the connection is Connect-PnPOnline. Enter it on a new row below the Import-Module command, and add the Url parameter and the path to the site that you copied.

```
Connect-PnPOnline -Url "https://kdemo.sharepoint.com"
```

For SharePoint server, the cmdlet is instead Connect-SPConfigurationDatabase and you can find info about the parameters at <https://docs.microsoft.com/en-us/powershell/module/sharepoint-server/connect-spconfigurationdatabase?view=sharepoint-ps>

Now you have a piece of script that imports the SharePointPnPPowerShell module and connects PowerShell to a SharePoint site.

```
Import-Module -Name SharePointPnPPowerShellOnline
```

```
Connect-PnPOnline -Url "https://kdemo.sharepoint.com"
```

Run these two lines, and you will be asked to enter your SharePoint credentials.

When you run the script, you might get a warning about inappropriate verbs. You can ignore this warning.

3.2.1 Check Connection

To check if PowerShell is connected to the SharePoint site, you can type the cmdlet Get-PnPList at the command prompt and run it. Now all lists and libraries on the site will be displayed in the console pane – including hidden lists.

```
PS C:\windows\system32> get-pnplist
```

Title	Id	url
-----	---	---
appdata	ac7cdcfc4-b0e3-41bb-9c6f-ce283152a738	/sites/PowerShell/_catalogs/appdata
appfiles	d390a4fc-7d43-42e9-b61a-7974525ec0ea	/sites/PowerShell/_catalogs/appfiles
Composed Looks	a3f508af-a407-4abe-bf52-6bfc9c04c43	/sites/PowerShell/_catalogs/design
Converted Forms	40795872-2800-4e6a-b13f-170bcd0b4215	/sites/PowerShell/InConvertedForms
Documents	3d2a9a80-2e6a-410c-88dc-7038330c3835	/sites/PowerShell/Shared Documents
Events	835c4209-c5fc-4e37-a573-5a87b24fac8c	/sites/PowerShell/Lists/Events
Form Templates	b7e2552d-2d08-4c97-839a-97889ab533e3	/sites/PowerShell/Formservertemplates
List Template Gallery	8f97e08d-fb5f-45dd-98db-e423f2280b6	/sites/PowerShell/_catalogs/lt
Maintenance Log Library	4b55a30a-d1d8-4f4d-ae99-9a64111d3bde	/sites/PowerShell/_catalogs/MaintenanceLogs
Master Page Gallery	9b33b98c-783c-42ee-9b47-0e041629356b	/sites/PowerShell/_catalogs/masterpage

When you have made sure that the connection works properly, you should save the script that connects to SharePoint. Then you can use it at a starting point for new scripts that perform the tasks you actually want to use PowerShell for.

3.2.2 Comment and comment out

Hashtags are used for comments to the script that PowerShell should ignore when running it. You can use them for any comments you like to add. You can also use the hashtag to *comment out* code that no longer should be run.

The SharePointPnPPowerShell installation is only performed once, so when the installation is finished, you should comment on the installation by typing a hashtag and some explanatory text.

Also comment out the installation code itself by putting a hashtag in front of it. In PowerShell ISE such text becomes green.

```
#First time tasks
```

```
#Install-Module SharePointPnPPowerShellOnline
```

Both these lines will now be ignored by PowerShell when the script is run.

You can also comment out both lines with <# at the beginning and #> at the end of the code. This is useful when you want to comment out multiple lines.

```
<#First time tasks
```

```
Install-Module SharePointPnPPowerShellOnline#>
```

I recommend that you make it a habit to makes comment in your script with hashtags like this. That will make it easier for everyone – including yourself in a few months! – to understand the code.

Should you need to use a block of code that has been commented out, you can just remove the # again.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/010-Intro-PowerShell-SharePoint.htm>

Comment on the demo:

It is not always necessary to include the parameter name in the code as long as you include the parameter value, and in this demo I only type the parameter values. Therefore, the two lines of script above look like this instead in the demo:

```
Import-Module SharePointPnPPowerShellOnline
```

`Connect-PnPOnline "https://kdemo.sharepoint.com"`

In this book I wanted to be extra clear. Therefore, I have included the parameters so that you would understand how they work. I will explain more about these so-called unnamed values in chapter 5.

It is convenient to save the SharePoint site and the login credentials in the script, so that you don't have to enter them each time. That is what we will do in our first exercise.

3.3 SUMMARY

In this chapter, we have created our first PowerShell script. This script has installed the SharePointPnPPowerShell module on your computer and set up a connection between PowerShell and SharePoint. This is the basics, that you must do before you can start managing SharePoint with PowerShell.

When you have commented out the installation rows in the script, as mentioned above, you should still have two active rows in your script:

`Import-Module -Name SharePointPnPPowerShellOnline`

`Connect-PnPOnline -Url "https://kdemo.sharepoint.com"`

Save this script, because we will continue working with the same script in multiple exercises.

To be able to go back and look at your scripts, you can also save it as a new version after each of the exercises.

Now we will go over to the exercises!

EXERCISES

The rest of this book contains exercises, or examples, on how to create script. If you are new to PowerShell and coding, I recommend that you go through the exercises in order, because the steps build on each other. This means that code written in earlier exercises are used in, and necessary for, scripts that are executed later.

At the end of the book, you can find the code used in each chapter, and I will also add many blocks of code in the exercise chapters. However, I strongly recommend that you read my instructions and not just look at the code. If you want to write your own scripts later, it is important that you understand why things are done as they are in the exercises. It is also crucial to know what things are called in PowerShell.

Before we go over to the exercises, I want to gather some general information that I might be useful to have in one place.

THEORY

In the Theory sections, I continue to give general information about PowerShell and script creation, but I choose such knowledge that is relevant for and practiced in the exercise at hand. I advise you to study the theory sections carefully and go back to them if necessary.

STEPS

The exercise steps build on what you have learned in the previous chapters and in the theory sections. They explain in detail how each piece of script should be written, and I have tried my best to explain why each step is taken. Always use the steps in combination with the theory section in the same chapter.

I recommend that you create the example scripts yourself and not just read about them. Even if your organization does not need all these scripts, creating them will teach you a lot that you can make use of when you build your own scripts later.

That way you will learn to write PowerShell code, and you will discover a lot more about the tool than I have included in this book.

The commands that import the SharePointPnPPowerShell module and connect to a SharePoint site should be present in all scripts. I will not repeat these rows in the exercises. In all exercises but the first one, that part will look like this:

```
Import-Module -Name SharePointPnPPowerShellOnline
```

```
$URL = "https://kdemo.sharepoint.com"
```

```
Connect-PnPOnline $URL
```

If you are using the browser login method to save the login credentials, you must also add the `-UseWebLogin` parameter to the `Connect-PnPOnline` cmdlet in each script.

This will be explained in the next chapter. There I will also explain the `$URL` you see above, because in the script we created in chapter 3 we did not use it.

DEFINITIONS

It is important that you understand the concepts I use in this book, to better follow the exercises and to be able to explore your PowerShell knowledge beyond this book by searching information online and in other books.

I introduce PowerShell commands and concepts throughout this book, but here is an alphabetical overview that you can go back to if you discover that you have forgotten something:

Array: a type of variable, with a set of components (array elements).

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

The elements of the array are numbered sequentially, starting with 0, and you can access an element at the PowerShell command prompt by using its index number. *Refer to* chapter 9.

- **Class:** a blueprint that specifies the properties of certain objects. An object that is created from a class, is called an instance of that class. *Refer to* chapter 20.

```

Class CV{
    $FirstName

    $LastName

    $Department

    $Decision
}

```

- **Command:** executes an action and always starts with a cmdlet. *Refer to* Commands.
- **Cmdlet:** starts a command. Some cmdlets can be used alone as a command, but cmdlets are more often expanded and specified with other code. *Refer to* Cmdlets.

```
New-PnPList
```

- **Element:** a component in an array.
- **Function:** a kind of cmdlet that you create yourself. Give it a name and enter within curly brackets which action(s) the function should execute.

```

function Create-MyList($ListName, $ListURL){
    New-PnPList -Title $ListName -Template GenericList -Url $ListURL
}

```

Run the function by calling its name and giving the applicable values for its parameters, if there are any. *Refer to* chapter 6.

```
Create-MyList -ListName "Hello World 3" -ListURL "HelloWorld3"
```

- **Hashtable:** a collection of data, enclosed in curly brackets and preceded by @. Can be used for key-value pairs or properties. *Refer to* chapter 14.
- **If statement:** modifies code so that it is only run if a certain condition is met. *Refer to* chapter 10.
- **Key-value pairs:** used when you have an object – the key – and a specific value that you want to link to that key. The key-value pairs are contained in a hashtable. *Refer to* chapter 14.

```
@{"Title"=$CountryName}
```

- **Loop:** runs a set of commands a number of times. It can either run through an array as many times as there are elements or just repeat the code as many times as needed.

Loops are written in the same pattern as functions, with the loop type, a statement and an action within curly brackets.

```
foreach($Dep in $Departments){  
    Create-MyList -ListName $Dep -ListURL $Dep  
}
```

In this book we use two loops:

- o For-loop: runs a specified number of times, often used to increment the loop condition. *Refer to* chapter 7.
 - o Foreach loop: to access an array and process its content. *Refer to* chapter 9.
- **Method:** specifies an action that you can perform on an object. In this book we are using string methods, like Replace and ToLower. Methods must always be followed by a parenthesis. *Refer to* chapter 10.

```
$ListURL =$ListName.Replace(" ", "").ToLower()
```

- **Object:** PowerShell is an OOP (object-oriented programming) language. This means that it is built on objects and that nearly everything we work with in PowerShell is an object.

Objects have many different types of information associated with them. In PowerShell, this information is sometimes called members.

- **Operator:** a sign that affects the relation between objects in a command, such as =, -eq, +, -lt. *Refer to* Operators.
- **Parameter:** a cmdlet specification. Parameters always have a name and sometimes a value, and the value can be mandatory for the cmdlet. Add a space and a hyphen after a cmdlet to get suggestions on parameter names

in PowerShell ISE. Values are suggested when you add a space after the parameter name. *Refer to Parameters and Values.*



Property: an attribute that describes an object. When you add a dot after an object, you will get IntelliSense suggestions to select from in PowerShell ISE.



- **Type:** All variables in PowerShell is of a specific data type, such as string, integer or datetime. PowerShell sets the data type automatically when you create a new variable, but it is often useful to set the type explicitly. *Refer to chapter 10.*

```
[string] $ListName
```

- **Variable:** stores values that can be changed. Variables always have a dollar sign before the name and an equals sign before the value. *Refer to chapter 4.*

```
$URL = "https://www.kalmstrom.com"
```

Variables that work as input to a function act as parameters. Variables inside a class definition act as properties.

MEANING OF PARAMETER/ARGUMENT AND VARIABLE

If you search for information about PowerShell online, you will find that there is a certain confusion even on Microsoft pages, around how the words parameter, argument and variable are used. These are all containers that have

names and values, and hence the confusion.

Parameter and argument is the same kind of container, and I will use the word parameter in this book. A variable is another kind of container, but the confusion is of the same kind as with parameters/arguments: the word is used for the container name, the value or both.

Compare with the meaning of the word lunchbox:

"Fill the lunchbox" – the container.

"Bring the lunchbox" – probably the container and its content.

"Eat the lunchbox" – definitely the content!

In the same way, parameter/argument and variable are used when people are referring to its name, its value or both.

I will try to be clear in this book and use parameter name/parameter value and variable name/variable value when I think there might be a risk of misunderstanding. But please bear with me, if I join the confusion that exists out there! I hope that you will still understand what I mean.

CMDLET DIFFERENCES IN SHAREPOINT EDITIONS

I am using SharePoint Online for the images and example code in this book, but even if you use an on-premises edition of SharePoint you should be able to follow this book.

All cmdlets and other code that are *not* specific for the SharePointPnP PowerShell modules are the same for all SharePoint editions (and for all other PowerShell coding). There are however some differences in the PnP cmdlets, simply because the SharePoint on-premises editions are not built in the same way as the online edition.

The first part of the PnP specific cmdlet is usually the same for all SharePoint editions, but the second part can be different. When you have installed the module for the SharePoint edition you are using, you will find all the correct cmdlets in the PowerShell ISE command pane, as I described in 2.4.1.2.

In the exercises, I will only give the SharePoint Online options for the PnP

cmdlets. The book would be too extensive if I gave all the cmdlets and parameters for other SharePoint editions too.

Therefore, it will mean some extra work for you to search for the corresponding PnP cmdlet and its parameters if you are using another SharePoint edition. I hope you will still find my book useful!

4 AVOID THE LOGIN

To avoid having to enter your log in details every time you want to run a script, you can save credentials information in the PowerShell script. In this chapter, I will describe two ways to do that.

The easiest way is to just save the browser login, but if you intend to use PowerShell more extensively, I recommend that you instead save the credentials in the Windows Credential Manager. That is more secure, and with this method you can let scripts run on a schedule.

In this chapter, I will also explain how to save the site URL in a variable in the script, so that you don't have to enter it each time you are working with the same site. Even better, if you want to use the script on multiple sites, you must only change the URL manually in one place in the script.

4.1 THEORY

In this exercise we are using variables for the first time – but surely not the last. Variables are very useful in all automation. We will also have a look at the concepts *string* and *assignment operator*.

4.1.1 String

Quotation marks, "", indicates that the text within the quotation marks should be considered a string. A string is any series of characters that are meant to be interpreted literally by the script. In PowerShell, the strings are normally values.

A string contains a set of characters and can also have spaces and numbers. For example, both "carrot" and "Please, give me 10 carrots" are strings.

Instead of double quotation marks you can use single quotes, ". This is necessary when the string itself contains double quotes, and vice versa: use double quotes when the string contains single quotes.

When you write PowerShell code, you don't have to add quotation marks around values that consist of only one word, like carrot or CarrotCost, because PowerShell will interpret them as strings by default. It is however a

good habit to always add double or single quotes around a string to avoid problems.

4.1.2 Variables

A variable is like a container that holds different types of information. You can use variables for storing, modifying and retrieving any kind of data. PowerShell uses many automatic variables, but here I will discuss the variables you can create yourself.

We will use many variables in more elaborate scripts in this book. Storing information in variables is, in general, a good idea and a recommended best practice.

The value is set once in a variable, and then the script always uses that value until you change it. A variable can also contain an array of multiple values.

In this exercise we will use the variable `URL` as a value instead of the actual URL in the command that connects to SharePoint.

```
Connect-PnPOnline -Url $URL
```

We will also use a `Creds` variable for the SharePoint credentials.

4.1.2.1 Create and Declare a Variable

In PowerShell, a variable must begin with a dollar sign. Directly after `$` comes the variable name, which you decide yourself. To give the variable a name is to declare the variable.

Note that the variable name is the part after the dollar sign. The dollar sign is only there to tell PowerShell to read the variable's value. Therefore, in `$URL` the variable name is `URL`.

It is best practice to use variable names that only include alphanumeric characters and the underscore.

PowerShell variable names are not case-sensitive, so you cannot have two different variables with the names `$myVariable` and `$Myvariable`.

PowerShell will not be able to distinguish between them.

Many other programming languages require a separate variable declaration and even type specification, but that is not needed in PowerShell.

4.1.2.2 Assign the Variable a Value

You can create a variable and give it a value in the same step. The variable values can be strings, numbers, an instance of a class or even more than one value, called an array. (Both class and array will be explained later in this book.)

Note that PowerShell only creates a new variable if it doesn't exist. Otherwise, it assigns the specified value to the existing variable.

Enter the variable value after the name separated by space, an operator which most often is the equal sign, space, like this:

```
$URL = "https://kdemo.sharepoint.com"
```

The quotation marks around the URL above indicates that it should be considered a string.

PowerShell automatically chooses the data type for the variable when you first assign it a value, but it is also possible to specify what type of values a variable can contain. We will talk more about that in chapter 10.

4.1.2.3 Initialize a Variable

In many programming languages, all variable names and values are forgotten when the code stops running, so run time and design time are important concepts in such languages.

In PowerShell it is not like that. Once a line of code with a variable has been executed, the variable name and value stay in the memory for as long as the PowerShell command prompt or development environment is running – or, when you use PowerShell ISE, for as long as the session is open.

Therefore, when you run the line `$URL = "https://www.kalmstrom.com"`, the URL variable is declared (given a name and a space in the memory) and assigned (given a value).

Another way of expressing this is that I introduce the variable name and assign the variable a value so that PowerShell remembers it. That is called to *initialize* the variable.

When I create a variable, I usually run that line of code right away to initialize it. If you don't do that, you must include the variable row the first time you run the code that includes the variable name.

If you change the variable value, you must run the variable row again, to re-assign the variable the new value.

Another reason to initialize a variable in PowerShell ISE is to get IntelliSense for it. In later chapters we will assign a value to a variable and run it just to get the IntelliSense.

Initializing is not necessary if you run the whole script at once, but as said earlier, I recommend that you continuously run parts of the script to test it instead of waiting until the whole script is finished.

4.1.2.4 Assignment Operators

An assignment operator assigns a value to a variable. PowerShell has eight such assignment operators. They are always added after the variable name and a space, and the most common one is the equals sign, =.

After the operator comes another space and then the variable value, as shown above. There can also be multiple values, as we will see later.

In this book, I will only use the assignment operators = and +=, but please refer to Microsoft if you need to use operators that perform numeric operations on the values before the assignment:

https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_assignment_operator?view=powershell-5.1

Note that = is only used when you assign a value to a variable. In other PowerShell contexts, -eq is used as the equality operator.

This is something that separates PowerShell from other programming languages and a common source of mistakes for me: I use = for comparison instead of for assignment. T

4.1.2.5 Properties

PowerShell variables can have properties, which store data about the current object. When you add a dot after a dollar sign and the name of a variable, you will have IntelliSense suggestions to select from – if the variable has been initialized.



4.1.2.6 Ask with a Variable

If you want to check the value(s) of a variable in a specific context, you can select the dollar sign and the variable name and run it. You can also type and run this at the command prompt.

In both cases, the value(s) of the variable will be shown in the console pane.

When you type \$ and a variable name and add a property, you can ask questions about the variable. In the image below, I have added the Count property to ask about the number of objects for the variable AllCVFiles. As you see they are 270.

```
PS C:\Users\PeterKalmström> $AllCVFiles.Count  
270
```

You will see more examples later in this book.

4.1.2.7 See all Variables

To get a list of the variables used in your PowerShell session, including all the built-in ones, type Get-Variable at the command prompt. The variable names are then displayed in the console pane, with their values.

There are many automatic variables, but as the variables are sorted alphabetically you should be able to quickly find the ones you are looking for.

4.2 STORE URL IN VARIABLE

I recommend that you store the URL to the SharePoint site you are working with in a variable. If you put it on top in the script, you can easily find and change it when you want to run the same script on other sites.

4.2.1 Steps: Store URL in Variable

Create a variable with the name URL and give it the value of the URL to your SharePoint site. Then you only need to initialize it – and that's it!

1. Enter a dollar sign and the name URL.
2. Add the equals assignment operator.
3. Add a string with the URL to your SharePoint site.

```
$URL = "https://kdemo.sharepoint.com"
```

4. Run the variable row to initialize the variable.
5. Now, when you have this variable, you can change the Connect-PnPOnline command you created in chapter 3, so that it does not show the URL to the site but instead uses the variable.

```
Connect-PnPOnline -Url $URL
```

6. Before you save the script, I recommend that you run the Get-PnPList cmdlet again, to make sure you are connected, *refer to* Check Connection.

4.3 SAVE THE CREDENTIALS

To avoid having to log in to SharePoint every time you want to run a piece of script, you can save your credentials so that the PowerShell script can reach them. There are two ways of doing this, one easy and one more complicated but also more secure and useful.

4.3.1 Save the Browser Login

When you let the browser save your password, you can take advantage of that in PowerShell by adding the parameter UseWebLogin to the Connect-PnPOnline command.

```
Connect-PnPOnline -Url "https://kdemo.sharepoint.com" -UseWebLogin
```

or (better), with the URL variable described above:

```
$URL = "https://kdemo.sharepoint.com"
```

```
Connect-PnPOnline -Url $URL -UseWebLogin
```

4.3.2 Save in Windows Credential Manager

When you are working with multiple sites, you might want to spend some more time on saving your login credentials in the Windows Credential Manager. This method is very safe, as it saves the password in a secure string.

Another important reason to store the SharePoint credentials in Windows Credential Manager is automation. When you want to automate the running of PowerShell scripts that connect to SharePoint, *refer to* chapter 18, you must have the credentials in the Credential Manager.

You can see and edit the saved link and credentials in the Control panel >Credential Manager >Windows Credentials >Generic Credentials.

Manage your credentials

View and delete your saved logon information for websites, connected applications and networks.



Web Credentials



Windows Credentials

[Back up Credentials](#) [Restore Credentials](#)

Windows Credentials

[Add a Windows credential](#)

TERMSRV/192.168.1.70

Modified: 2019-08-16

Certificate-Based Credentials

[Add a certificate-based credential](#)

No certificates.

Generic Credentials

[Add a generic credential](#)

Microsoft_ExpressionWeb_kalmstrom\kalmstromadmi...

Modified: 2019-08-23

PnPPS:https://kalmstromnet.sharepoint.com/sites/de...

Modified: Today

Internet or network address:

PnPPS:https://kalmstromnet.sharepoint.com/sites/demosite/

User name: kate@kalmstrom.com

Password: *****

Persistence: Enterprise

[Edit](#) [Remove](#)

4.3.2.1 Steps

We will first create a variable that stores the credentials, and then we will add this credential variable to Windows Credential Manager.

1. Create a variable named Creds.
2. Assign the cmdlet Get-Credential as a value to the variable.

```
$Creds = Get-Credential
```

3. Run the variable row to initialize the variable.
4. Enter your username and password in the dialog that opens and click OK.
5. Now your password will be saved as a secure string in the Creds variable.
You can check the value by selecting and running just \$Creds or run it at the command prompt.

```
PS C:\Users\PeterKalmström> $Creds
UserName                                     Password
-----                                     -
peter@m365x135719.onmicrosoft.com system.Security.SecureString
```

6. To add the credentials to Windows Credential Manager, enter the cmdlet Add-PnPStoredCredential on a new row.
 1. Add the parameter Name and the value \$URL, to have the site URL.
 2. Add the parameter Username and the value \$Creds. Add a dot after Creds and select the property Username. This takes the username from the Creds variable.
 3. Add the parameter Password and the value \$Creds with the property Password. This takes the password from the Creds variable.
- ```
Add-PnPStoredCredential -Name $URL -Username $Creds.UserName -Password $Creds.Password
```
4. Run the Add-PnPStoredCredential command to save the credentials from the Creds variable to Windows Credential Manager.
  5. Check in the Credential Manager that your login details to SharePoint are properly saved.

6. Once the credentials have been stored and saved in a correct way, you don't have to repeat the process in each new script. Therefore, the rows above can be commented out and moved to the first time installation part of the script.

Now the first part of the script looks like this:

```
#First time tasks
#Install-Module SharePointPnPPowerShellOnline
#$Creds = get-credential
#Add-PnPStoredCredential -Name $URL -Username $Creds.UserName -Password $Creds.Password

import-Module SharePointPnPPowerShellOnline

$URL = "https://kdemo.sharepoint.com"

Connect-PnPOnline -Url $URL
```

Follow these steps if you want to run the script on another site collection:

1. Change the value of the URL variable.
2. Initialize the variable to assign the new value.
3. Remove the hashtag on the Add-PnPStoredCredential command.
4. Run the Add-PnPStoredCredential command to change site collection in Windows Credential Manager.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/020-Save-Login-PowerShell-SharePoint.htm>

## 4.4 SUMMARY

In this chapter we have used a variable for the first time, but certainly not for the last! We have seen how variables are created, declared, assigned a value and initialized.

In the exercises, we stored the SharePoint URL in a variable and saved the SharePoint credentials to Windows Credential Manager.

Finally, I explained what you must do if you want to work with another site collection than the one that was specified in the URL variable and in the Credential Manager.

Now it is time to start managing SharePoint with PowerShell. In the next chapter, we will build a command that creates a list app and another command that removes it again.

## ***5 CREATE AND REMOVE A SHAREPOINT APP***

I assume that you already know how to create and remove a single app from the SharePoint UI, but if you use PowerShell you can create multiple apps of the same kind in very short time and remove apps just as quickly. You just need to run a PowerShell script!

In this exercise, we will use the cmdlet `New-PnPList` to create a new SharePoint list app. We will only create one list this time, but in the next chapter we will create ten lists at the same time.

We will also create a command that removes the app again.

### **5.1 THEORY**

Just as in the previous chapter, this exercise requires a rather heavy theory input. Don't feel discouraged, because the concepts I am introducing here are necessary for the coding and will come back several times later in the book. So, you will have a chance to try this again!

#### **5.1.1 Apps = Lists**

In PowerShell, all apps, both regular lists and libraries, are called lists. You select what kind of app that should be created when you choose template in the command.

In this exercise, we will use the value `GenericList` for the `Template` parameter. This will give us a modern list app with just a title column.

```
New-PnPList -Title "Hello World" -Template GenericList
```

We will use the same cmdlet and template value in several chapters ahead, but we will modify the command and its context a bit in each chapter to make it more useful.

If you want to create another kind of app, just change the `Template` parameter value into something that suits you better. In the next section, I will explain how to find the value.

#### **5.1.2 Command Pane and IntelliSense**

As I explained in section 2.4.1.2, the PowerShell ISE command panes are useful when you want to find the correct cmdlet and its parameters.

You can also add parameter values in the command panes.



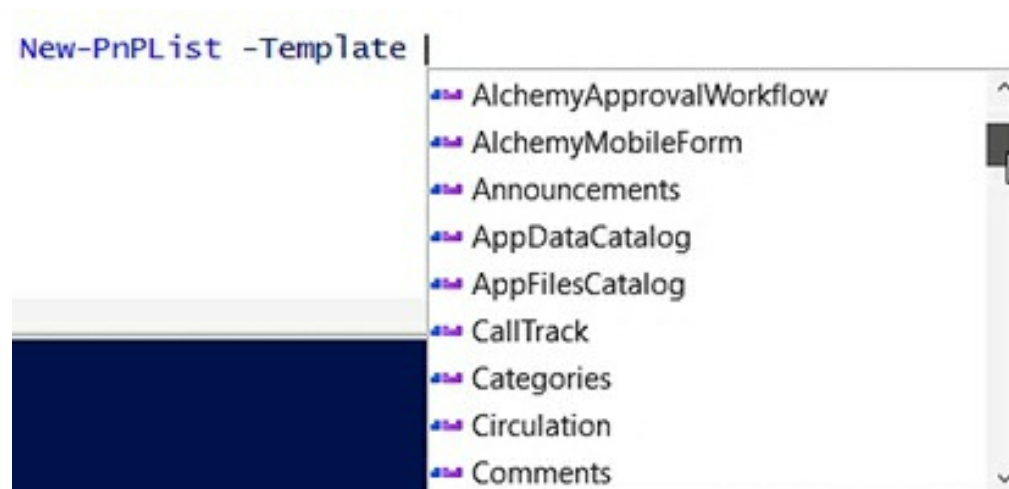
For example, Template is a mandatory parameter for the New-PnPList cmdlet we use in this exercise. If you know which value you want to use for the Template parameter in the NewPnPList command, you can add it in the command pane together with other parameters and values.

When you click on the Run button, the new list will be created in SharePoint. The only problem is that your command will be added to the console pane, where it cannot be saved.

But if you don't know the correct value for the app you want to create, you will have another problem first: you will not get any help in the command panes, because they have no IntelliSense.

To have IntelliSense, you should instead use the script pane. Here you can find the right value among the IntelliSense suggestions.

Add a space after the Template parameter to get the value suggestions.



As PowerShell works in this way, I prefer to just look at the options and the information in the command pane and then type everything in the script pane. There I will have IntelliSense, and I can save my script and use it more times.

### 5.1.3 Unnamed Parameter Values

The default parameter for a value in a PowerShell command does not have to be written out in the script. Also, if there is only one valid parameter for a value, you don't have to write out the parameter name in the command.

The Remove-PnPList cmdlet, which we will also use in the exercise, has one default parameter, Identity. Its value should be the list ID or title.

Therefore, the command

```
Remove-PnPList -Identity "Hello World" -Force
```

can be written without the Identity parameter name, as

```
Remove-PnPList "Hello World" -Force
```

The values of such hidden parameters are called unnamed values. When we remove the list app we will create in this exercise, we will use an unnamed parameter value.

There are however reasons for always supplying the parameter names: the code becomes more readable, and it is a security when using functions, *refer to the next chapter*. Therefore, I only use unnamed parameter values when there is no risk for misunderstandings or errors.

## 5.2 CREATE APP

In this exercise we will create a new SharePoint generic list with the name Hello World. To get a good-looking URL without special characters, we will use the Url parameter and give it a value that has no spaces: HelloWorld.

We will also put a link on the Quick launch and enable versioning in the list app. This is done with two parameters that do not take any values: OnQuickLaunch and EnableVersioning.

Such parameters are also called *switches*. A switch can either be present or not present. The OnQuickLaunch switch (which of course also is a parameter) can either be present or not present when you use the New-



PnPList cmdlet. If it is there, a permanent link to the newly created list will be added in the Quick launch navigation of the site where it is created.

## 5.2.1 Steps

When I write "Enter", start on a new row. When I write "Add", continue on the same row.

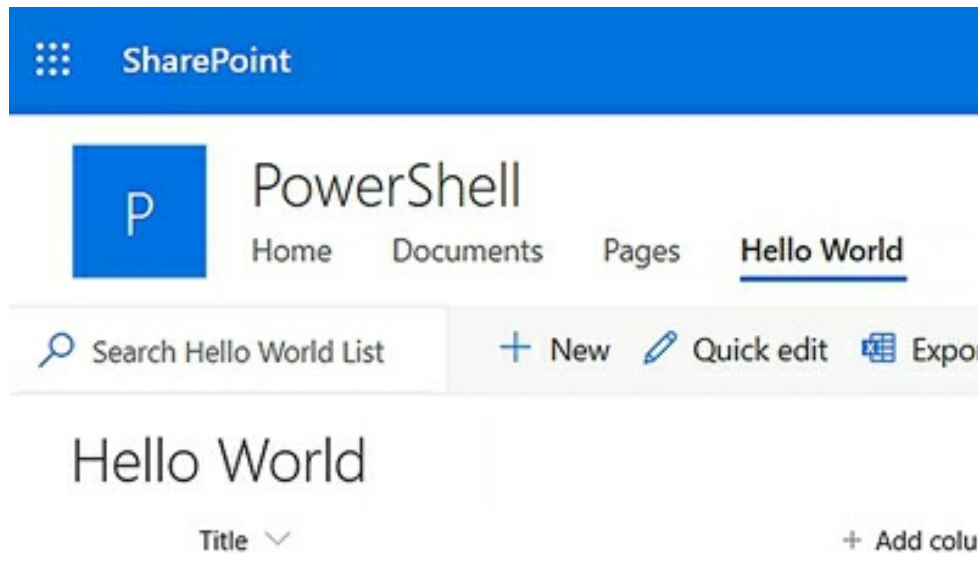
1. Enter the cmdlet New-PnPList.
2. Add the Title parameter name with the parameter value of the string Hello World.
3. Add the Template parameter with the value GenericList.

```
New-PnPList -Title "Hello World" -Template GenericList
```

4. Add the parameters/switches EnableVersioning and OnQuickLaunch.
5. Add the Url parameter and its value, the string HelloWorld.

```
New-PnPList -Title "Hello World" -Template GenericList -EnableVersioning -OnQuickLaunch -
Url "HelloWorld"
```

6. Run the New-PnPList command.
  1. Check that the new app has been created in the SharePoint site PowerShell is connected to, that a link is added to the Quick Launch and that versioning is enabled in the List settings. Also, check that the URL to the newly created list is indeed created without %20 for the space between the words Hello and Word.



Note that I have added the app to a modern Communication site in the image above, so the Quick launch is placed on top of the page.

## 5.3 REMOVE APP

Here we will create a simple line of code that removes a SharePoint app, so that you can try the app creation and then quickly remove the app so that you can try again.

### 5.3.1 Steps

I prefer having the removal command at the bottom of the script, to minimize the risk of running it by mistake.

1. Enter the cmdlet Remove-PnPList.
2. Add a string with the name of the app as an unnamed value.
3. To avoid having to confirm removal of the app, add the Force parameter.

```
Remove-PnPList "Hello World" -Force
```

4. Run the command and check in SharePoint that the app has been removed.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/030-Create-Remove-App-PowerShell-SharePoint.htm>

## 5.4 SUMMARY

In this chapter, we have used the cmdlet `Add-PnPList` to create an app in SharePoint. I have explained how to find parameters in the PowerShell command panes and how you can get value suggestions for them by using the PowerShell IntelliSense.

We also used the `Remove-PnPList` cmdlet to delete the app we created from SharePoint. In the `Remove-PnPList` command we did not write out the parameter. Instead we used an unnamed value.

As a SharePoint manager you will create a lot of apps, so the following chapters will show you more advanced ways to create apps from PowerShell. In the next chapter, we will build a function for app creation.

## ***6 FUNCTION FOR APP CREATION***

In the previous chapter, we wrote a PowerShell command that creates a SharePoint app. In this chapter, we will expand that piece of script to make it more generic and possible to use for multiple apps.

You could of course copy and paste the New-PnPList command we created and then change the relevant values for each new app, but using a function is a better and more generic way. In a later chapter we will make such a function work for any kind of SharePoint app.

Even if it takes some time to create a function, it will be well worth the effort, and you will not only thank yourself for it – your colleagues and anyone else who takes over the script from you will also feel grateful that you used a function!

In this chapter, we will still only create one app, but in the next chapter we will let a for-loop run through the function we create in this chapter. With a function and a loop, you can create as many apps as you want by just running a piece of script.

### **6.1 THEORY**

In this exercise we will use variables, so please look back on chapter 4 if you don't remember the theory behind a variable. I will also introduce the PowerShell function and the cmdlet Write-Host.

#### **6.1.1 Functions**

A function is a kind of cmdlet that you create yourself. We use functions for blocks of code that we want to run several times.

When commands are contained within a function, it is easy to re-use them. To execute the commands inside the function, you only need to call the function – enter its name – and maybe add some parameters and their values.

Another reason for building a function is to encapsulate complexity. For example, it is a good idea to implement the (very sensible) business rule that all lists should have versioning in a function. This is done by adding the

EnableVersioning parameter name to the function.

If you want to have new apps added to the Quick launch, it is also a good idea to add the OnQuickLaunch parameter name to the function.

#### **6.1.1.1 Create a Function – Name**

Start each function with the word function and a name, for example function Send-Greeting.

You should try to find a good, descriptive name for your function, not only for your own sake but also to make it easier for other people to take over your script.

The recommended way is to write the function name as "Verb-Noun", just like the built-in cmdlets. You can check if the name is already in use by running Help [Name] at the command prompt, where [Name] is the word you want to use in your function.

You can also generate a list of the approved PowerShell verbs by running the cmdlet Get-Verb.

A function is similar to a cmdlet in several ways. The main difference is that you create it yourself and must specify each step that should be taken. (As mentioned earlier, the built-in cmdlets most often have hidden commands encapsulated.)

#### **6.1.1.2 Create a Function - Commands**

The function commands are written inside curly { } brackets. You can have just one command in a function, but there can also be a long sequence of commands that make the function very complex. The commands are executed in same order as the rows where they are written.

Everything inside the curly brackets should be tab indented, for better readability. For the same reason, you should also keep the commands on separate rows, and they should not be placed on the same row as the start and end curly brackets.

The start curly bracket is placed at the end of the row that starts with function and the name, and the end bracket is written on a separate row.

It might look like this:

```
function Create-MyList{
 New-PnPList -Title "Hello World List" -Template GenericList -Url "HelloWorld"
}
```

Within the curly brackets, you should also place other building blocks that modify or affect the commands. Such a building block can be an If statement (chapter 10) or a foreach loop (chapter 25) or a variable (chapter 28).

### 6.1.1.3 Create a Function - Input Parameters

After the function name, we can add a parenthesis with any number of input parameters. These are often variables that are used as parameters in the function calls, *see* below. Add a comma between the input parameters.

```
function Create-MyList($ListName, $ListURL){
```

When we call the function, we use the variable names as parameter names, and that is why we call them input parameters.

Input parameters are optional, and the first function above has no such parameters. Sometimes the parenthesis is written out even if it is empty, but that is not necessary.

### 6.1.1.4 Initialize a Function

Functions need to be initialized in the same way as variables, so if you want to run just a part of a script that has a function, you must run only the function first. Select the whole function (including the end curly bracket) and press F8 or click on the Run Selection icon.

### 6.1.1.5 Call a Function

The only time when you run a function is when you initialize it after creation or changes. To get a result from the function, you must call it, not run it. When you call the function, the command(s) within the curly brackets will be executed.

Call the function by entering its name. Often, we also need to add the input parameters and the values that should be used in this particular execution of the function.

When we do that, we use the variable names as parameter names, and we have the hyphen that signifies a parameter before the name, like this: -

URL.

```
Create-MyList -ListName "Hello World 3" -ListURL "HelloWorld3"
```

As you see, the call looks like when we are using a cmdlet command, and that is also what the call is. The difference is that you have created this cmdlet yourself. When you use a function name to call a function, it will therefore get the same color as the built-in cmdlets in PowerShell ISE.

You must enter the call below the function itself in the script. If you write the call above the function, it will not work, as PowerShell executes all code in the order it is written from top to bottom.

You can either type the function call in the script pane somewhere below the function or enter it at the command prompt. Which option you choose, depends on if the call should be kept in the script or not. Calls that should be kept are often written inside other blocks of code. We will see several examples on that later in the book.

As we saw in the previous chapter, parameters do not always have to be written out. This function call without parameter names would work too, as well as the one you see above:

```
Create-MyList "Hello World 3" "HelloWorld3"
```

However, when several unnamed values are sent to a function, they will be assigned to the function parameters in the order they are written, and that can create problems if the function later will get a third parameter between the earlier ones.

Therefore, if you want to ensure that the function call will still work as intended if the function parameters would change, you should always name parameter values and write out the parameters as in the first function call above.

### 6.1.1.6 Expand and Collapse Functions

Functions can be collapsed in the PowerShell ISE script pane, by the small plus and minus icons to the left of the function itself. This is the expanded version of the function we will create in this chapter.

```
function Create-MyList($ListName, $ListURL){
```

```
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
```

```
$ListURL
```

```
Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

And here is the collapsed version:

```
function Create-MyList($ListName, $ListURL){...}
```

As you see, the lines of code within the curly brackets are shown as three dots only, so it is a good idea to collapse functions you are not working with, to save space in long scripts and make your code more readable.

Later in this book, I will introduce loops and If statements. They also have code between curly brackets, and they can be collapsed and expanded in the same way as functions.

## 6.1.2 Write-Host

The cmdlet Write-Host tells PowerShell to give a message in the console pane. In this exercise, we will use it to give a progress message after the New-PnPList command has been executed. Therefore, the Write-Host command must be written below the row with the New-PnPList command (but still within the function's curly brackets), *see* above.

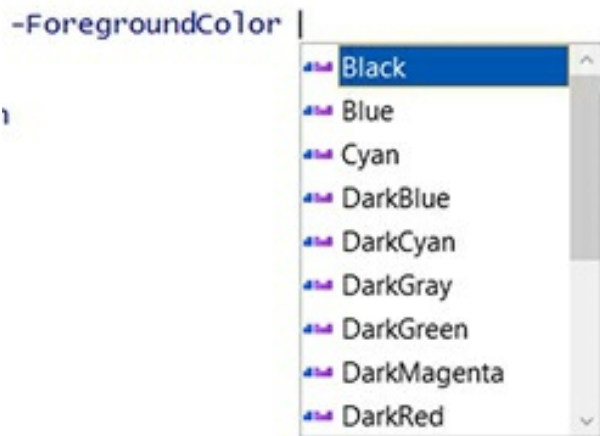
The message is displayed in the console pane.



```
PS C:\Users\PeterKalmström> Create-MyList -ListName "Hello World 3" -ListURL "Helloworld3"
List Hello World 3 has been created
```

The text in the console pane is white by default, but for better visibility, we will use the parameter ForegroundColor. This parameter gives the message text the color that we define in its value, in this case green.





For white text on a colored background, use the parameter `BackgroundColor` instead.

`Write-Host` takes the default parameter `Object` for the message that should be displayed, but with this cmdlet I usually don't find it necessary to write out the parameter. Instead I use an unnamed value for the console pane message.

However, it is of course also possible to name the parameter and write `Write-Host -Object` and then a parenthesis with the message that should be displayed in the console pane.

### 6.1.2.1 Addition Operator and Parenthesis

In the `Write-Host` command, we will concatenate the different parts of the progress message with the addition operator, a plus sign, and the whole message to be displayed is given in a parenthesis.

```
Write-Host ("List " + $ListName + " has been created!")
```

The parenthesis is there to indicate that (just like in math) whatever is inside the parenthesis should be calculated first. The result of the calculation becomes the (here unnamed) value of the `Object` parameter that is sent to the `Write-Host` cmdlet.

## 6.2 STEPS

I have divided this exercise in three parts: create the function, call it and add the progress message.

We will use the `New-PnPList` command we created in the previous exercise,

but we will replace the Title and Url parameters with variables.

When we call the function, the variable names will act as parameters that we can give any values we wish. That way, we can use the function to create apps with different names.

### 6.2.1 Create a Function

Here we will create a simple function that adds a generic list app to SharePoint.

1. Enter a new function with the name Create-MyList.
2. Copy the New-PnPList command from the previous chapter and paste it inside the function's curly brackets.

```
function Create-MyList{
 New-PnPList -Title "Hello World List" -Template GenericList -EnableVersioning -
 OnQuickLaunch -Url "HelloWorld"
}
```

3. Add the variables ListName and ListURL as input parameters after the function name.

```
function Create-MyList($ListName, $ListURL){
```

4. Inside the curly brackets, replace the values of the Title and Url parameters with the variables.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -Url $ListURL
}
```

5. Initialize the function.

### 6.2.2 Call the Function

Now we will test the function by calling it. We will use the input parameters and give them the values we want for the app this time.

When we have checked that the list has been created, we will remove it again.

1. Enter the name of the function: Create-MyList.
2. Add the two input parameters and their values for this call. In this case the list name is Hello World 3 and the URL is the same but without spaces.
3. `Create-MyList -ListName "Hello World 3" -ListURL "HelloWorld3"`
4. Run the Create-MyList call.
5. Now the function will take the ListName and ListURL parameter values that you entered in the call and create the list app using those values.
6. Check in SharePoint that the Hello World 3 list has been created in the SharePoint site that PowerShell is connected to and that the URL is correct, without spaces.
7. In the Remove-PnPList command from chapter 5, change the Identity value to the one we used to create this new list: Hello World 3.
8. Run the Remove-PnPList command to remove the list again, and check that it is gone from SharePoint.

### 6.2.3 Add Progress Message

We will now expand the function by adding a Write-Host command. It will add a message to the PowerShell ISE console pane that the list has been created.

1. Inside the function's curly bracket, after the New-PnPList command, enter the cmdlet Write-Host.
2. Within a parenthesis, add the word List as a string, the variable ListName and the string "has been created". Use the addition operator to concatenate the three parts of the message.
3. Add the parameter ForegroundColor with the value Green.

```
Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
```

4. Now the whole function looks like this:

```
function Create-MyList($ListName, $ListURL){
```

```
New-PnPList -Title $ListName -Template GenericList -Url $ListURL
```

```
Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

5. Call the function again. As you removed the earlier list, you can use the same call as above.
6. Check that the progress message is displayed in the console pane and that a new list has been created in SharePoint.



```
PS C:\Users\PeterKaleström> Create-MyList -ListName "Hello World 3" -ListURL "HelloWorld3"
List Hello World 3 has been created
```

7. Use the Remove-PnPList command to remove the list.

Demo: <https://www.kalmstrom.com/Tips/PowerShell-SharePoint/040-First-Function-PowerShell-SharePoint.htm>

## 6.3 SUMMARY

In this chapter, I have introduced the PowerShell function, and you have now seen how to create a function with input parameters and some commands. You also understand how to execute a function by calling it.

Write-Host was the new cmdlet in this chapter, and we used it to make PowerShell show a progress message in the console pane. The message was created within a parenthesis, and an addition operator concatenated its three parts.

In the next chapter, I will show how you can use a loop to create multiple list apps from the function we have created.

## ***7 CREATE MULTIPLE APPS WITH FOR-LOOP***

In this chapter we will continue the list app creation, and now we will create ten different lists with the same settings but with different names. For that, we will use a so called "for-loop" that calls the function we created in the previous chapter.

We will also remove these ten lists with another for-loop that uses the Remove-PnPList cmdlet.

### **7.1 THEORY**

This chapter explains what a for-loop is and how it is written and used, and I also introduce two operators that are included in the for-loop we will build.

#### **7.1.1 Less Than and Increase Operators**

In the for-loop we create in this exercise, we use the -lt operator. -lt means less than, and there is of course also an operator called -gt (greater than).

In the condition statement for our for-loop, this operator will give us a number of loops that is less than 11 – which means ten loops.

`$i -lt 11`

We also use the operator ++, which increases the value of the variable with one for each loop.

`$i++`

Both these operators can of course be used in other contexts in PowerShell as well.

#### **7.1.2 For-Loop**

PowerShell has several loops that are very useful. In this exercise we will use two so called for-loops. These loops are often used to run a command or a set of commands a specified number of times, and they often use an integer variable. Such a variable is usually written as \$i.

Here we will use a for-loop to call the function we created in the previous

chapter. The loop will run ten times, to create ten list apps in SharePoint. The second for-loop will run the Remove-PnPList command ten times to remove the lists again.

A for-loop is built in a similar way as a function. It starts with 'for' and then comes the statements for the loop with inside a parenthesis. These statements most often consist of a variable with different operators, separated with a semi-colon.

```
for($i=1;$i -lt 11;$i++)
```

The parenthesis above contains these statements in this order:

1. The value of the variable when entering the loop. In our example:  $\$i = 1$ .
2. The condition on which the loop should be run. In our example, when the variable value is less than 11:  $\$i -lt 11$ . As long as that evaluates to true, the command within the curly brackets will be executed, the counter variable  $\$i$  will be incremented and so on.
3. An action that will be performed against the variable value each time the for-loop runs. In our example, increase the value with 1:  $\$i++$

After the statements, and within curly brackets, comes the command(s), in this case the Create-MyList call.

Thus, the for-loop checks the statements and then executes the block of code within the curly brackets repeatedly, for as long as the second statement, the condition, is met.

```
for($i=1;$i -lt 11;$i++){
 Create-MyList -ListName ("Hello World" + $i) -ListURL ("HelloWorld" + $i)
}
```

## 7.2 LIST CREATION

Here we will create a for-loop that runs ten times through the function we built in chapter 6 and creates ten list apps in SharePoint.

### 7.2.1 Steps

Before you start with this script, make sure that you have removed the list in SharePoint that we created in the previous chapter.

1. Enter for and a parenthesis to start creating the for-loop.
2. Inside the parenthesis, add the first statement, the variable i, for integer, with the value 1, followed by a semicolon.
3. Add the condition to continue looping as long as the current variable value is less than eleven, followed by a semicolon.
4. Add a statement, that increases the value with 1 for each loop.
5. Add the curly brackets.

```
for($i=1;$i -lt 11;$i++){
}
```

6. Enter a Create-MyList function call within the for-loop's curly brackets. Use the list name Hello World.

```
for($i=1;$i -lt 11;$i++){
 Create-MyList -ListName "Hello World" -ListURL "HelloWorld"
}
```

7. Add parentheses around the ListName and ListURL value strings.
8. Add the addition operator and the integer variable inside the parentheses.

Because of the parentheses, the values inside them will be calculated before the ListName and ListURL values are sent to the Create-MyList cmdlet.

```
for($i=1;$i -lt 11;$i++){
 Create-MyList -ListName ("Hello World" + $i) -ListURL ("HelloWorld" + $i)
}
```

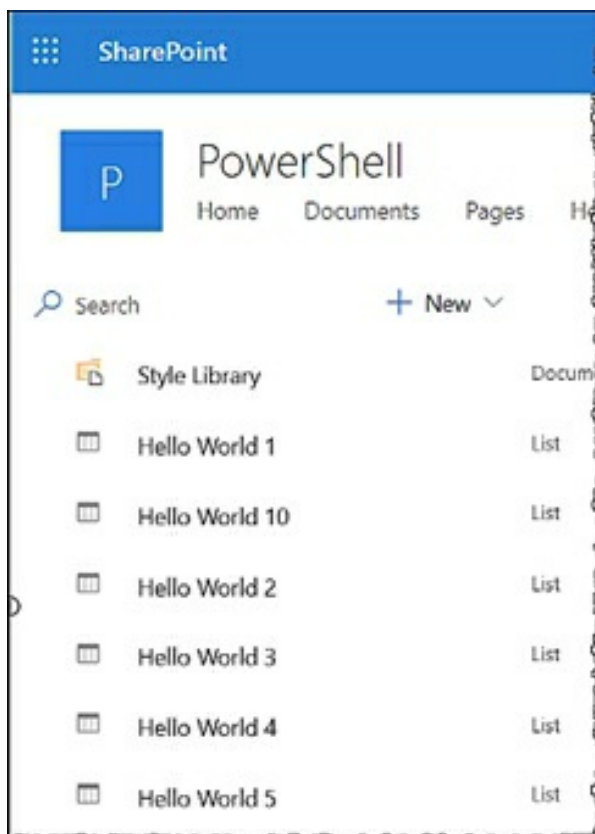
9. (Run the function, to make sure it is initialized.)
10. Run the whole for-loop to call the function and create the ten lists.

```
Windows PowerShell
Connect-PnPOnline $URL
10
11 function Create-MyList($ListName, $ListURL){
12 #Always versioning on
13 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url $ListURL
14 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
15 }
16
17
18
19 -for($i=1;$i -lt 11;$i++){
20 Create-MyList -ListName ("Hello world " + $i) -ListURL ("helloworld" + $i)
21 }
22
23
24
25
26
27 Remove-PnPList "Hello world" -Force

New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url $ListURL
Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}

PS C:\Users\PeterKaleström> for($i=1;$i -lt 11;$i++){
Create-MyList -ListName ("Hello world " + $i) -ListURL ("helloworld" + $i)
}
List Hello world 1 has been created!
List Hello world 2 has been created!
List Hello world 3 has been created!
List Hello world 4 has been created!
List Hello world 5 has been created!
List Hello world 6 has been created!
```

11. Check that you can see all the lists in SharePoint.





## 7.3 LIST REMOVAL FOR-LOOP

To remove the lists, we will use the for-loop we created in the previous steps, but with a Remove-PnPList command inside the curly brackets.

### 7.3.1 Steps

Here I will copy and paste, but to practise, you can of course also create a new for-loop at the bottom of the script pane and give it the Remove-PnPList command.

1. Copy the whole for-loop we created above and paste it at the bottom of the script pane.
2. Inside the curly brackets, enter the Remove-PnPList cmdlet.
3. Add the unnamed value of the Identity parameter within a parenthesis. The value should be the string Hello World with the addition operator and the integer variable.
4. Add a -Force parameter to avoid ten confirmation messages when you run the removal command.

```
5. for($i=1;$i -lt 11;$i++){
 Remove-PnPList ("Hello World" + $i) -Force
}
```

6. Select the for-loop and run it.
7. Check in SharePoint that all the Hello World lists have been removed and moved to the recycle bin.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/050-For-Loop-PowerShell-SharePoint.htm>

## 7.4 SUMMARY

An integer variable is often used in a for-loop that runs a command a

specified number of times. In this chapter, we have used such a for-loop to call the Create-MyList function that we created earlier ten times, to add ten list apps to SharePoint.

We have also created a for-loop that removes the same apps again.

In the next chapter, I will show another way of creating multiple apps in a short time, by using Excel to build function calls.

## 8 CODE CREATION IN EXCEL - MULTIPLE CREATE-MYLIST CALLS

If you have an Excel list with names that you want to use as app names in SharePoint, you can quickly create a function call code inside Excel. Then you just need to copy the code, paste it into PowerShell ISE and run it, to have the SharePoint apps created.

In this chapter and the next, I will show two different ways to create such code. In both exercises, we will create a command that will call the function we created in chapter 6.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -Url $ListURL
 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

These two chapters require some basic knowledge of Excel. If you don't have that, you should still study the PowerShell code in these exercises, because we will build on them in later chapters.

I recommend my book *Excel 2016 from Scratch*, which is available from Amazon, if you want to learn more about Excel.

### 8.1 PREREQUISITES

In this and the next exercise, we will use an Excel list with department names, and we want each of these departments to have a list app built on the generic list template. Each of the apps should have the same name as the department that will use it.

The department names have no spaces, so the department name can be used in the URL without problems. (If not, you now know how to make them appear without spaces!)

Even if we use a list of department names here, you can of course use any Excel list with names that you want to use as app names. However, the Excel list should be *formatted as a table* for this to work as described below.

## 8.2 THEORY

When an Excel list is formatted as a table, you can fill down a formula from the first row, and the correct values will be added to the following rows. This is something we will take advantage of in the first method that creates code for PowerShell in Excel.

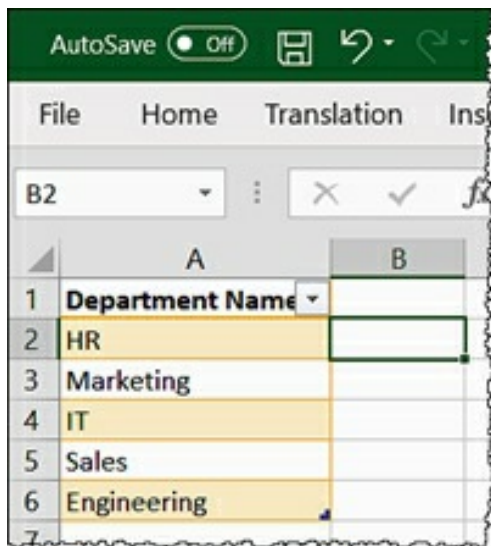
What we will do in Excel here, is simply to create a function call like the one we have used earlier, but we will create one call for each department.

We only need to create the call for the first department, because then we can drag it down and get the code for the other departments.

It is important to know that formulas in Excel must start with = and that & is used to concatenate strings in Excel, just as + is used in PowerShell.

## 8.3 STEPS

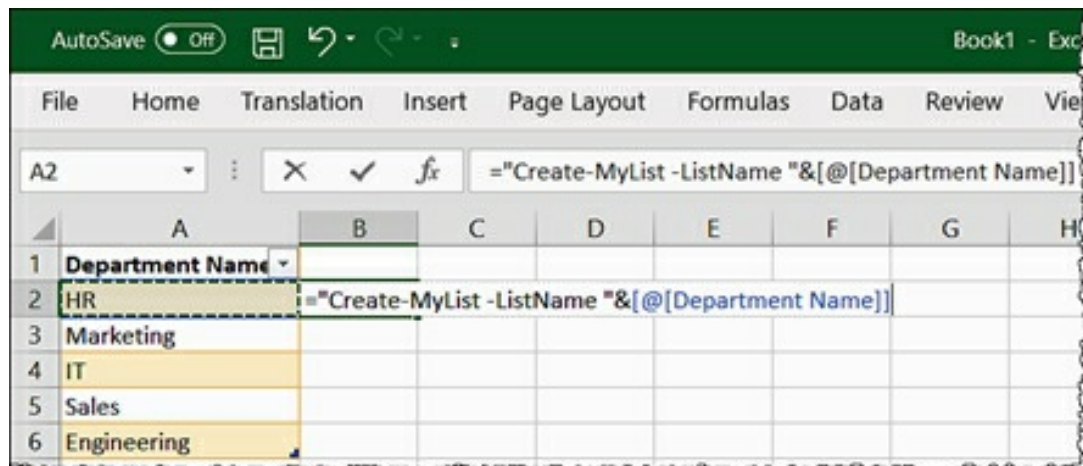
We will start entering the formula in the cell that is selected in the image below: B2.



1. Enter = and then " to start building a string.
2. Add the function name, Create-MyList and the ListName parameter, either by typing or by copy and paste from the PowerShell script we have already created.

3. Add a space and " to finish the string. The space must be there, because the PowerShell function call requires a space between the ListName parameter and its value.
4. Add &.
5. Select the cell with the first department name (A2 in the image above). Now that part of the formula will be entered automatically:

= "Create-MyList -ListName "&[@[Department Name]]



6. Add a space and "&" to start a new string.
  7. Add the ListURL parameter and space. (the space here is also for the PowerShell code requirements, as explained in point 3).
  8. Finish the string.
  9. Enter & and select the first department name cell again to have the department name in the list URL.
- = "Create-MyList -ListName "&[@[Department Name]] &"-ListURL "&[@[Department Name]]
10. Press Enter on the keyboard to have the formula filled out in the whole column. It will now have the department names in the first column filled out as text strings.

| Department Name | Column1                                              |
|-----------------|------------------------------------------------------|
| HR              | Create-MyList -ListName HR -ListURL HR               |
| Marketing       | Create-MyList -ListName Marketing -ListURL Marketing |
| IT              | Create-MyList -ListName IT -ListURL IT               |
| Sales           | Create-MyList -ListName Sales -ListURL Sales         |
| Engineering     | ListURL *&[@[Department Name]]                       |

11. Copy the generated function calls, B2 to B6 in the image above.
12. Paste the code you copied from Excel below the Create-MyList function in PowerShell ISE.
13. Select all the calls and run them. They will now call the Create-MyList function in the order they are entered and create the lists with the department names.

```

24
25 Create-MyList -ListName HR -ListURL HR
26 Create-MyList -ListName Marketing -ListURL Marketing
27 Create-MyList -ListName IT -ListURL IT
28 Create-MyList -ListName Sales -ListURL Sales
29 Create-MyList -ListName Engineering -ListURL Engineering
30

```

```

List HR has been created!
List Marketing has been created!
List IT has been created!
List Sales has been created!
List Engineering has been created!
PS C:\Users\PeterKalmström>

```

14. When the script run is completed, check in the Site contents that the new apps have been created in SharePoint.

## 8.4 REMOVE APPS

To remove the apps we just created, we will use Excel to create Remove-PnPList commands.

Start in the same Excel row as above, in the first empty cell to the right of the department name in the table.

1. In Excel enter "=" and the cmdlet Remove-PnPList.
2. Enter a space and finish the string.
3. Enter &.
4. Select the cell with the first department name. Now that part of the code will be entered automatically.

= "Remove-PnPList "&[@[Department Name]]

5. Press Enter on the keyboard to have the formula filled out in the whole column with the department names as text strings.

|   | A               | B                                                        | C                          |
|---|-----------------|----------------------------------------------------------|----------------------------|
| 1 | Department Name | Create Code                                              | Remove Code                |
| 2 | HR              | Create-MyList -ListName HR -ListURL HR                   | Remove-PnPList HR          |
| 3 | Marketing       | Create-MyList -ListName Marketing -ListURL Marketing     | Remove-PnPList Marketing   |
| 4 | IT              | Create-MyList -ListName IT -ListURL IT                   | Remove-PnPList IT          |
| 5 | Sales           | Create-MyList -ListName Sales -ListURL Sales             | Remove-PnPList Sales       |
| 6 | Engineering     | Create-MyList -ListName Engineering -ListURL Engineering | Remove-PnPList Engineering |

6. Copy the generated commands.
7. Paste the commands you copied from Excel at the bottom of the script pane in PowerShell ISE.
8. Select all the Remove-PnPList commands and run them.
9. Check that all the lists with the department names have been removed from SharePoint.

Demo: <https://www.kalmstrom.com/Tips/PowerShell-SharePoint/060-Excel-Code-PowerShell-SharePoint.htm> – the first part

## 8.5 SUMMARY

Excel can be used to create code for PowerShell in a quicker way than if we type it in manually. In my example in this chapter I have only used five departments, but imagine if you have many more! You will save a lot of time by using Excel.

In this exercise we have used Excel to create the same kind of function calls that we used in chapter 6, one call for each department.

This chapter had no new PowerShell theory, but in the next chapter I will introduce new, important building blocks again. We will use them to create SharePoint apps with department names, just as we did here, but you will learn a more elegant way that will be useful in many other contexts too.



# ***9 CODE CREATION IN EXCEL - CONTENT STRING***

In this chapter we are using the same function and Excel sheet as in the previous chapter, and we will create SharePoint lists here also – but we will use different methods.

Another way to create blocks of code in Excel, is to join the departments in a content string that will be used to create an array in PowerShell. This array will then be used in a foreach loop that runs through the Create-MyList function and creates the lists.

## **9.1 THEORY**

Now I will introduce two very important PowerShell building blocks that we are going to use much more in the following chapters: the array and the foreach loop.

I will also explain how the PowerShell indexing works, and for better understanding of this exercise I will include a piece of Excel theory.

### **9.1.1 Array**

An array is a variable value that consists of a set of components, called array elements. A delimiter that you decide yourself separates each element. Here we are using an array with five elements in the form of strings. The strings are separated by a comma delimiter.

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

This chapter's method to create PowerShell code in Excel uses such an array, and the elements in the array are the department names we get from the Excel table. We will use single quotation marks for the five strings of department names.

The array elements can also be fetched from another data source by a cmdlet. We will use such arrays later in this book, the first time in chapter 15.

When you have created a variable with an array value in PowerShell, you can

use the variable for questions in the console pane just like you can do with other variables.

In the image below, I have used the variable with the name Departments that you can see above. If I just run the dollar sign and the name at the command prompt, I will get the array elements listed.

When I add the property Count, I can see how many elements there are in the array.

```
PS C:\Users\PeterKalmström> $Departments
HR
Marketing
IT
Sales
Engineering

PS C:\Users\PeterKalmström> $Departments.Count
5
```

### 9.1.1.1 PowerShell Indexing

The indexing in PowerShell starts with zero. Therefore, when I enter the Departments variable and add [0] and then run it, the result is the HR department – which we might consider to be number one in the array!

For the same reason, the third department in the array is Sales and not IT.

```
PS C:\Users\PeterKalmström> $Departments[3]
Sales

PS C:\Users\PeterKalmström> $Departments[0]
HR
```

### 9.1.2 TEXTJOIN

To create the array for this exercise, we will use the Excel function TEXTJOIN, that joins text from two or more strings together. In this case, the function will join the five department names.

| Department Name | Create |
|-----------------|--------|
| HR              | Create |
| Marketing       | Create |
| IT              | Create |
| Sales           | Create |
| Engineering     | Create |

`=TEXTJOIN(",",",",TRUE;[Department Name])`

In Excel, the comma delimiter is written as `","`. The double quotation marks are for the string, and the single quotes and the comma are for the delimiter.

The TEXTJOIN function has a parameter for ignoring empty cells, `ignore empty`, and we will set it to `TRUE` in this exercise.

### 9.1.3 Foreach Loop

In chapter 7, we created a for-loop that can run through a function, and in this exercise, we will use a foreach loop. This is another kind of loop that is most often used with an array.

The foreach loop starts with the word `foreach`. After that comes a parenthesis that includes a statement with two variables and the word `in`.

```
foreach($Dep in $Departments){
}
```

The first variable represents the current array element that the loop is running through at that moment (here `$Dep`). We don't have to set a value for this variable, as this is done automatically by PowerShell each time the foreach loop runs through the array.

In our example, the foreach loop will run five times, as the `Departments` array has five elements. The first time the loop runs, the `Dep` variable gets the value of the first element, `HR`. In the second run, the `Dep` variable gets the value of the second element, `Marketing` – and so on.

After the variable for the current element comes `"in"` and then the array variable (here `$Departments`), which represents all the array elements.

Within the curly brackets we place the command that tell what the foreach loop should do with each element in the array. That line of code should be

indented for best readability.

More advanced foreach loops than the one we use in this chapter, contain multiple commands. We will create such foreach loops later in this book.

Other building blocks that affect the command(s) in the foreach loop should also be placed within the curly brackets. It can be a variable (*see* chapter 13), an If statement (*see* chapter 17) or even another foreach loop (*see* chapter 23).

When you use a foreach loop to call a function, the loop executes the function commands on each element in the array. In this exercise, the foreach loop calls the Create-MyList function five times – because there are five elements in the array.

Let's have a look at the function's New-PnPList command:

```
New-PnPList -Title $ListName -Template GenericList -Url $ListURL
```

Compare it with the call in the foreach loop:

```
Create-MyList -ListName $Dep -ListURL $Dep
```

As you see, the function variables ListName and ListURL are used as parameters in the foreach loop and they have the value of the current department.

This way, the foreach loop tells the function to use the current department name as value for the function's variables ListName and ListURL when creating the lists. And, as the current department name is different for each loop, we will have five different list names and URL values.

## 9.2 STEPS

We will start in Excel and create a string from the five departments. The string will then be used to create a PowerShell array, which will be used in a foreach loop that calls the Create MyApp function for each element in the array.

### 9.2.1 Create a String

To create the content string in Excel, start in the cell below the last department name, A7 in the image below.

|   | A               |
|---|-----------------|
| 1 | Department Name |
| 2 | HR              |
| 3 | Marketing       |
| 4 | IT              |
| 5 | Sales           |
| 6 | Engineering     |
| 7 |                 |

1. Enter =TEXTJOIN
2. Add a start parenthesis.
3. Start a string.
4. After the double quotation mark, add ',' to specify the delimiter between the department names in the array.
5. End the string.

=TEXTJOIN(",")

6. Add semicolon before the next parameter and set "ignore empty" to TRUE, so that any empty cells will be ignored.

|   |                                                        |    |
|---|--------------------------------------------------------|----|
| 6 | Engineering                                            | Cr |
| 7 | =TEXTJOIN(",",";TRUE                                   |    |
| 8 | TEXTJOIN(delimiter; ignore empty; text1; [text2]; ...) |    |
| 9 |                                                        |    |

7. Add another semicolon before the next parameter.
8. Drag the cells in the department column down to the code cell, A7, to get all the department names into the function.
9. End the parenthesis.

=TEXTJOIN(",",";TRUE;[Department Name])

10. Press Enter to finish the content string. [Department Name] will now be replaced with the actual department names.
11. Copy the value from the A7 cell.

## 9.2.2 Create an Array

Now we are going to use the string from Excel to create an array in PowerShell.

1. Create a variable called Departments.
2. As the variable value, paste the content string you copied from Excel.
3. Add single quotation marks before HR and after Engineering, to get five strings in the array.

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

4. Initialize the variable.

## 9.2.3 Use the Array in a Foreach Loop

We will use the array in a foreach loop. The loop will call the Create-MyApp function for each element in the array.

1. Enter foreach() on a new line below the Departments array.
2. Inside the parenthesis, add the foreach loop statement: \$Dep in \$Departments.

```
foreach($Dep in $Departments)
```

3. Add curly brackets.
4. Add the function name on a new, indented row after the first curly bracket.
5. Add the ListName parameter and set its value to the current array element.
6. Enter the ListURL parameter and set its value to the current array element. (In this case the list name and URL values can be the same, as the department names have no spaces.)

```
foreach($Dep in $Departments){
 Create-MyList -ListName $Dep -ListURL $Dep
}
```

7. Run the foreach loop.
8. In the SharePoint Site contents, check that the apps with the department

names have been created.

## 9.3 REMOVE APPS

To remove the apps we just created, we can create a similar foreach loop with a Remove-PnPList command inside the curly brackets.

The removal should of course be done with each array element - \$Dep – and we don't want to have any confirmation messages.

```
foreach($Dep in $Departments){
 Remove-PnPList $Dep -Force
}
```

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/070-Excel-To-Create-Arrays-Foreach-PowerShell-SharePoint.htm>

## 9.4 SUMMARY

We will use arrays and foreach loops many more times in this book, so this chapter was just an introduction.

We created two foreach loops that run through an array. The command in the first loop is a call to the Create-MyList function that we have created earlier. The second loop uses a Remove-PnPList command.

To not complicate things, I used department names without spaces in this chapter, so that they could be used in the URL values without creating bad looking URLs. In the next chapter, we will however enhance the URL value so that it never uses spaces and always uses minor letters.

## ***10 FUNCTION ENHANCEMENTS***

In this chapter, we will enhance the app creation function we have created earlier, so that the URL for each app – if it has not been supplied already – always is written without spaces and always has lower case letters.

With this enhancement, an app with the name Hello World will have helloworld in the URL.

```
https://m365x135719.sharepoint.com/sites/PowerShell/helloworld/AllItems.aspx
```

### **10.1 THEORY**

I introduced variables in chapter 4, and after that we have used variables in most of the exercises. In this chapter, we will see how you can restrict a variable to only contain data of a specific type.

We will also look at the If statement, that sets conditions for the execution of a script, and we will use string methods to work with strings.

#### **10.1.1 Type a Variable**

PowerShell variables support many data types, and if you don't do anything about it, PowerShell will automatically choose a data type when you first give the variable a value.

However, in some cases PowerShell will not use the data type you intended. It might for example happen that PowerShell considers numbers as text, and when that occurs the variable cannot be used for calculations.

Therefore, it is a good habit to set the data type when you create a variable. When that is done, PowerShell will try to convert any data that is used with the variable into the specified data type. If that is not possible, PowerShell will give an error message.

For example, almost everything can be converted into a string, but the string "Peter Kalmström" cannot be converted into an integer.

To set the type for a variable, add the data type within square brackets before the variable. For example, [Int] \$Number specifies that the value of the



Number variable must be an integer.

Other useful data types are [String], [Array], [DateTime] and [Bool] (which means that the value is either true or false).

In the exercise below we will use the string data type for the ListName and ListURL variables.

```
function Create-MyList([string] $ListName, [string] $ListURL){
```

When a variable name is used as a parameter name in a command, it will keep the string that the variable was typed with. That means that the value of the parameter can only be a string, if the variable was typed as a string.

### 10.1.2 If Statement

When you write code in PowerShell, you sometimes want the script to decide if an action should be taken or not. In such cases, you can use an If statement.

The If statement starts with If. After that comes a parenthesis with the actual statement, which can be evaluated to either true or false. Comparing a variable value to an expected literal value is a common way of doing this.

If the statement evaluates to true, the actions within the following curly brackets will be executed. If the statement evaluates to false, PowerShell will skip over all commands within the curly brackets after the If statement.

A very simple If statement might look like this:

```
if($Sum -eq 10){
 Write-Host("The sum is 10")
}
```

This means that if the value of the variable Sum equals (-eq) 10, the script will write "The sum is 10" to the console pane. If the value is not 10, nothing will be written to the console pane.

An Else statement can be used together with the If statement, to specify the action that should be taken when the If statement is false.

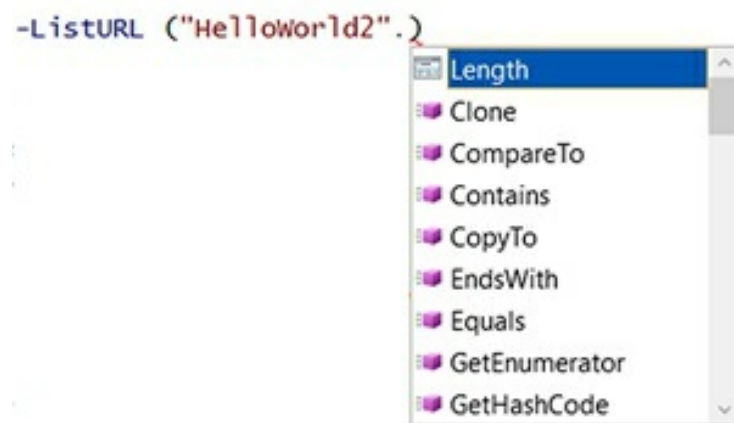
### 10.1.3 Methods

All PowerShell objects have methods that specify an action you can perform

on them. Here we will work with string methods and use them for the current variable object.

You can for example want to display the result in lower case. That method is ToLower. ToUpper and Replace are other methods, and I will give more examples later in this book.

When you add a dot immediately after a string in PowerShell ISE, you will have a choice of string methods.



You will have the same string method choices for variables that you have typed as strings and for variables that PowerShell considers to be strings.

Methods can have parameters, and these are put inside parenthesis and separated by a comma. If there are no parameters, you must still type the parenthesis.

In this exercise the method Replace has two parameters, the string that should be replaced (a string with a space) and the text that should replace it (an empty string). The method ToLower has no parameters.

```
$ListURL =$ListName.Replace(" ","").ToLower()
```

In the command above, we first use the method Replace to take away spaces from a string. Then we take the resulting string from the Replace method and converts it to lower case. The result of these two methods is put into the variable \$ListURL.

For example, if the variable \$ListName contains the string value “2020 Invoices” the \$ListURL variable will be given the string value “2020invoices” after these two methods are called.

## 10.2 STEPS

We will first type the variables in the function, and then we will add the If statement. Finally, we will call the function to test it, and we will of course use a list name with capital letters and a space.

### 10.2.1 Type the Function Variables

The function we have used earlier looks like this:

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -Url $ListURL

 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

Now we will enhance this function by setting the data type for the variables. We will declare that the variables ListName and ListURL should be of the data type string.

1. Add the variable type [string] before the two variables inside the parenthesis right after the function name.

```
function Create-MyList([string] $ListName, [string] $ListURL){
```

2. Run the function again, to initialize the changes.

### 10.2.2 If Statement

Here we will add an If statement to the function. It will check if the ListURL variable has a value or not.

If the value is null, we will set the ListURL to the same value as the ListName but with methods that remove any spaces in the list name and make the URL use lower case letters in the list name.

1. Enter If after the start curly bracket in the function.
2. Add a statement that compares the ListURL variable value to an empty string.

```
if($ListURL -eq "")
```

3. Add curly brackets.
4. After the start curly bracket, specify that the ListURL variable should have the value of the ListName variable.
5. Add a dot and select the string method Replace.
6. Within the method parenthesis, add the two parameters a string that contains a space and an empty string. Separate them with a comma.
7. Add a dot and select the string method ToLower.

```
if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ","").ToLower()
}
```

### 10.2.3 Call the Function

Now the whole function with the If statement looks like this:

```
function Create-MyList([string] $ListName, [string] $ListURL){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ","").ToLower()
 }
 New-PnPList -Title $ListName -Template GenericList -Url $ListURL
 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

1. On a new row below the function, enter the function name Create-MyList to call the function.
2. Add the ListName value "Hello World".

```
Create-MyList "Hello World"
```

Note that the function's second input parameter, ListUR, was not sent to the function in this call. In PowerShell, function input parameters are optional by default and you will not get an error when you call functions like this.

3. Run the call and check if the Hello World list is created in SharePoint and has a URL with the list name in lower case and without a space.

You can also use one of the function calls we created in chapter 6, but in that case, you can remove the ListURL parameter name and value from the call. Now we specified the value of the ListURL variable in the If statement.

4. Run the Removal foreach loop and check that the list has been removed.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/075-Typing-Variables-If-PowerShell-SharePoint.htm>

## **10.3 SUMMARY**

Maybe you don't think that the look of an URL is important, but a nice-looking URL gives a professional impression and is easier to remember.

However, my most important reason for including URL enhancements in this chapter was another: I wanted to introduce several PowerShell concepts in a context I hope is easy to understand.

Now you should know how an If statement is used, what typing a variable means, and how you can select and use string methods.

The next chapter is short and easy – we will just create a function that creates a SharePoint document library in the same way as we created a generic list.

# ***11 CREATE DOCUMENT LIBRARIES***

This chapter has very little new theory, but the exercise is still important as a preparation for the general app function that we will create in next chapter.

In this exercise, we will create libraries for the array of five department names that we extracted from the Excel file in chapter 9.

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

We will use the function we enhanced in the previous chapter as a starting point and make some changes to adapt it to a SharePoint document library.

I have chosen to use the foreach loop that we also created earlier to call the function.

## **11.1 THEORY**

This is what distinguishes a function that creates SharePoint list apps from a function that creates SharePoint document libraries:

- The function name should reflect that libraries are created.
- The value of the Template parameter must be DocumentLibrary.
- The progress message should reflect that a library has been created.

## **11.2 STEPS**

We will start the library function from the list function we already have, and the first step is to copy the list function and paste the copy further down in the script.

(There is a reason why I want you to copy and not just change the existing function. You will understand that in next chapter.)

The new function will create a SharePoint document library instead of the earlier generic list. After that, we must also change the foreach loop so that it calls the new function.

1. Use the function name Create-MyLibrary.

2. The value of the Template parameter must be DocumentLibrary.
3. The first string in the progress message should have the word Library.









```
function Create-MyLibrary([string] $ListName, [string] $ListURL){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template DocumentLibrary -Url $ListURL

 Write-Host ("Library " + $ListName + " has been created!") -ForegroundColor Green
}
```

4. Initialize the new function.
5. Change the foreach loop so that it calls the Create-MyLibrary function.

```
foreach($Dep in $Departments){
 Create-MyLibrary -ListName $Dep
}
```

6. Run the foreach loop and check that the progress message is correct and that the libraries have been created in SharePoint.

|                                                                                   |                |                  |
|-----------------------------------------------------------------------------------|----------------|------------------|
| <b>Contents</b>                                                                   | Subsites       |                  |
| <hr/>                                                                             |                |                  |
|  | Name           | Type             |
|  | Documents      | Document library |
|  | Engineering    | Document library |
|  | Form Templates | Document library |
|  | HR             | Document library |
|  | IT             | Document library |
|  | Marketing      | Document library |
|  | Sales          | Document library |

7. Run the Removal foreach loop and check that the library has been removed.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/080-Create-Libraries-More-Functions-PowerShell-SharePoint.htm>, the first two minutes.

## 11.3 SUMMARY

This exercise was just a preparation for chapter 12, that is rather heavy. Now you can create a document library and a generic list from PowerShell.

However, if you create many different app types, you need something better. It is not very efficient to use a separate function for each app type.

In the next chapter, we will solve that problem and create a PowerShell function that can be used for any kind of app.



## ***12 CREATE A GENERAL APP FUNCTION***

So far, we have been working with functions that create generic lists and document libraries, but with just a few changes such a function can be used for any kind of SharePoint app.

We will build a Create-MyApp function and call it with a foreach loop that runs through the departments array from chapter 9.

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

Users might prefer to keep the specific functions for generic lists and document libraries that we have already created. Therefore, we will also change these two functions so that they call the general app function when you run them. This way, any future changes of parameters or values must only be done in one function, even if you have three of them.

### **12.1 FUNCTION THAT CREATES ANY APP**

When you want to create a general function for all SharePoint apps, instead of hardcoding the app type in each function, you need to make changes in the function as well as in the call.

#### **12.1.1 Theory**

To create a general function for creation of all kinds of SharePoint apps, we need to add a general app type variable in the function and enter a specific app type parameter in the function call.

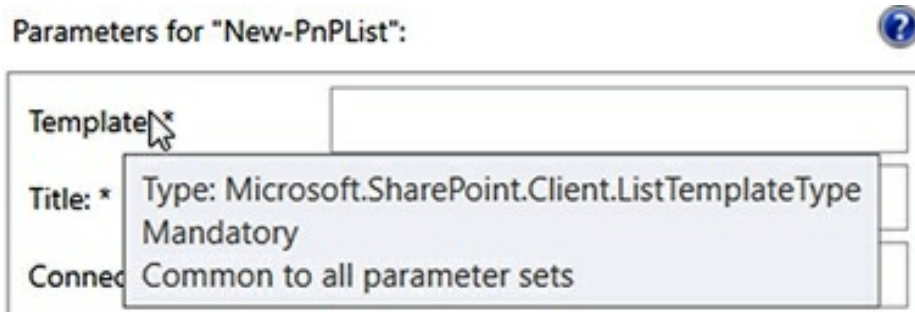
##### **12.1.1.1 Specify the App Type with a General Type**

The general app function needs the variable AppType, and it should be typed with a Microsoft type for all SharePoint lists/apps.

```
function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
```

To find this type, we will open a command pane and select the SharePointPnP PowerShell module and the cmdlet New-PnPList.

Now parameters for that cmdlet will be displayed, and when you hover over the Template entry you will see which type you should enter to include all app types: Microsoft.SharePoint.Client.ListTemplateType



Start writing the type after a start square bracket before the AppType variable, and you will have suggestions for the completion.



### 12.1.1.2 Convert the AppType Parameter to a String

We will use the AppType variable in the function's progress message, but it cannot be used with the type Microsoft.SharePoint.Client.ListTemplateType. Therefore, we need to convert the AppType variable into a string, with the method ToString. When that is done, we can use it in the progress message.

Now we also need to add a space string before the ListName variable.

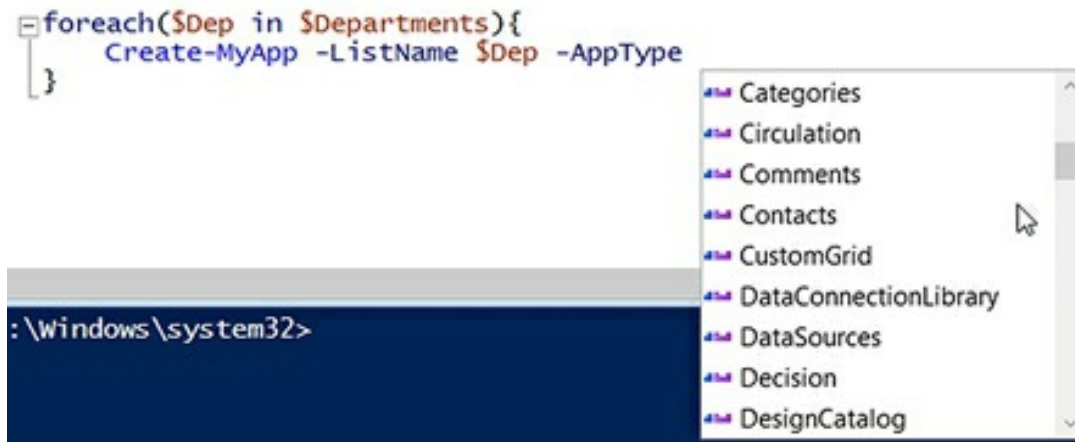
```
Write-Host ($AppType.ToString() + " "
```

### 12.1.1.3 Call a Function for All App Types

When we call a function for all app types, we must set the specific app type in the function call instead of in the function. Do that by adding the AppType parameter to the call and selecting the template as its value.

```
foreach($Dep in $Departments){
 Create-MyApp -ListName $Dep -AppType Tasks
}
```

You will have suggestions for the AppType value.



### 12.1.2 Steps

Now we will create a function that can be used for all kinds of SharePoint apps. Start with copying the original list function, so that we can work from that copy.

1. Change the function name into Create-MyApp.
2. After the ListURL variable in the input parameters parenthesis, add the variable AppType.
3. Type the AppType variable with Microsoft.SharePoint.Client.ListTemplateType.

```
function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
```

4. In the New-PnPList command, replace the value of the Template parameter with the AppType variable.

```
New-PnPList -Title $ListName -Template $AppType -Url $ListURL
```

5. In the progress message, replace the original output string "Library" with the AppType variable. Add a dot after the AppType variable and select the method ToString.
6. Add + " " after the ToString parenthesis, to add a space between the app type and the app name.

```
Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
```

7. Now you should have a function that looks like this:

```
function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template $AppType -Url $ListURL
```

```
Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
}
```

8. Initialize the new function.

9. Copy the foreach loop (that we created in chapter 9) and paste it below the end curly bracket of the original foreach.

10. In the copy, change the call to Create-MyApp and add the parameter AppType. Select the Tasks template as the value of the AppType parameter.

```
foreach($Dep in $Departments){
 Create-MyApp -ListName $Dep -AppType Tasks
}
```

11. Run the foreach loop and check that the progress messages look correct and that tasks lists for the five departments have been created in SharePoint.

12. Use the removal foreach loop to remove the lists again.

## 12.2 CALL FROM FUNCTION TO FUNCTION

It might be valuable for less experienced users to have dedicated functions for list and library creation, because that will spare them the trouble of selecting template.

Therefore, it is not obvious that you should delete the functions for generic list and library that we created earlier, even if we now have the general app function.

Instead, you can make the two dedicated functions for generic list and library call the general app function that we created above.

We will add the AppType parameter and its specific values in the two functions, but we can also remove a lot of code from them that now is specified in the general function.

If we later want to change any parameters or values, we can do that in the general app function.

### 12.2.1 Steps

Here I will give the steps for the Create-MyList function. For the library function, *see below*.

1. After the function's start curly bracket, enter the name of the general function, Create-MyApp.
2. Add the ListName and ListURL parameters and the variables that are their values.
3. Add the AppType parameter and set its value to GenericList.
4. Remove the If statement and everything below it except the end curly bracket, because these parameters are stated in the Create-MyApp function and are no longer needed here.
5. Now the Create-MyList function looks like this:

```
6. function Create-MyList([string] $ListName, [string] $ListURL){
 Create-MyApp -ListName $ListName -ListURL $ListURL -AppType GenericList
```

```
}
```

7. Initialize the new Create-MyList function.
8. Call the Create-MyList function with the Create-MyList foreach loop that we have used earlier.
9. 

```
foreach($Dep in $Departments){
 Create-MyList -ListName $Dep
}
```

10. Check that the SharePoint lists have been created.
11. Use the removal foreach loop to remove the lists again.

Follow the same steps to modify the Create-MyLibrary function to call Create-MyApp, but point 3 should be "Add the AppType parameter and set its value to DocumentLibrary".

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/080-Create-Libraries-More-Functions-PowerShell-SharePoint.htm> – start after two minutes

## 12.3 SUMMARY

The general function for app creation that we have built in this chapter did not give any new concepts. Instead, I hope you have seen how the building blocks you have learned so far can be used for really useful PowerShell scripts. Now, you can create SharePoint apps of different types in a short time.

In the next chapter, you will learn how to create new site collections from PowerShell.

## ***13 CREATE AND REMOVE SITE COLLECTIONS***

In the previous chapter, we took the final step in app creation with PowerShell and created a function that can be used to generate any kind of SharePoint app within the current site. Now we will go over to site collection creation.

### **13.1 CREATE SITE COLLECTION WITH COMMUNICATION SITE**

In this exercise, we will write a foreach loop that runs through the array of department names that we have used in several exercises already.

This time, the foreach loop will create a site collection for each department. The root sites will have the same names as the departments.

|             |                       |
|-------------|-----------------------|
| Engineering | .../sites/Engineering |
| HR          | .../sites/HR          |
| IT          | .../sites/IT          |
| Marketing   | .../sites/Marketing   |
| Sales       | .../sites/Sales       |

The sites we create in this chapter are each of them the root site in a new site collection and of the type modern Communication site.

Only the root site will be created in this exercise. In chapter 22, I will show how to create subsites.

#### **13.1.1 Theory**

In this theory section, I will explain how information in the command pane

can help you build the correct command for site collection creation. I will also show how you can use the Get-Help cmdlet and get help in the console pane.

### 13.1.1.1 Select Cmdlet from Command Pane

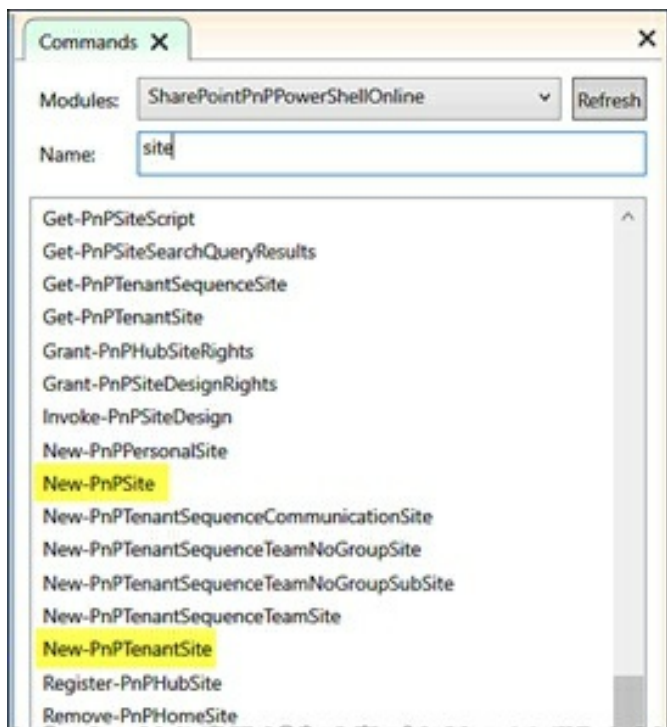
The command add-on and command window are very useful for finding the correct cmdlet and its parameters, even if you then type them in the script pane and don't insert them in the console pane. I discussed that in the Theory section of chapter 5.

Here we will use a command pane again, this time for finding the correct cmdlets and parameters for site collection creation and removal.

When you want to create a site that should become the root site of a new site collection, there are two possible cmdlets for SharePoint Online: New-PnPSite and New-PnPtenantSite.

The New-PnPSite cmdlet can only create modern sites, like Communication sites and modern Team sites.


If you want to create a classic site collection, you must use the New-PnPtenantSite option, but this cmdlet can also create modern sites.





When we look at the required parameters for the two cmdlets, we see that the New-PnPTenantSite has four mandatory parameters.

Name: New-PnPTenantSite  
Module: SharePointPnPPowerShellOnline (Imported)

Parameters for "New-PnPTenantSite": 


|             |                      |
|-------------|----------------------|
| Owner: *    | <input type="text"/> |
| TimeZone: * | <input type="text"/> |
| Title: *    | <input type="text"/> |
| Url: *      | <input type="text"/> |

Template or Type is not one of the mandatory parameters, and if you don't add a template parameter, this cmdlet will create a classic site collection. It is however possible to create modern sites with the New-PnPTenantSite also. You just need to provide the correct template name.

You can get all template names by running the cmdlet Get-PnPWebTemplates at the command prompt.

The New-PnPSite cmdlet only requires one parameter: Type.

Name: New-PnPSite  
Module: SharePointPnPPowerShellOnline (Imported)

Parameters for "New-PnPSite": 

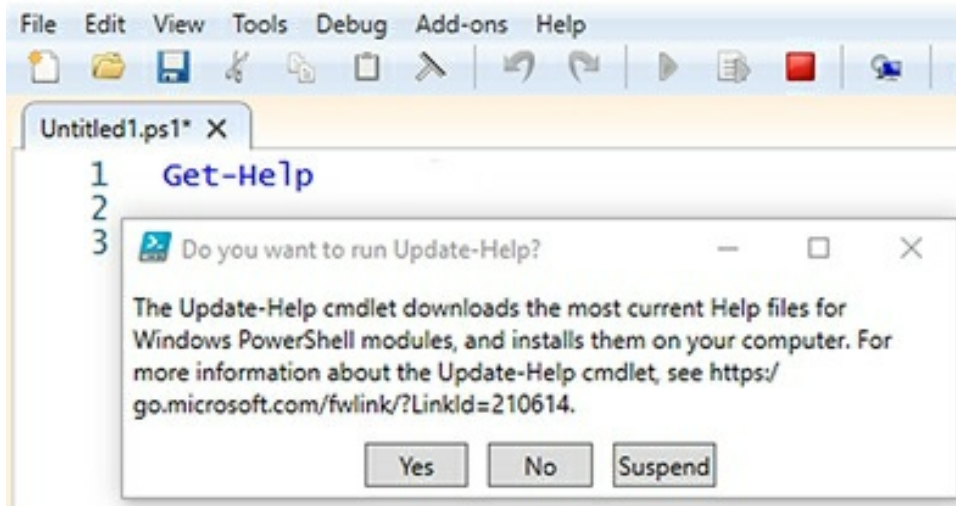
|             |                      |
|-------------|----------------------|
| Type: *     | <input type="text"/> |
| Connection: | <input type="text"/> |
| HubSiteId:  | <input type="text"/> |

We want to specify Type, Title and Url for the Communication sites, but we do not require Owner or TimeZone. Therefore, I have selected to use the New-PnPSite cmdlet for the exercise we are doing in this chapter.

For removals, there is only the cmdlet Remove-PnPTenantSite, and the only mandatory parameter is Url.

### 13.1.1.2 Get-Help

When you need help with a cmdlet in PowerShell, you can write Get-Help at the command prompt. The first time you run the cmdlet Get-Help, you will be asked to install the PowerShell Help files on your computer, under Windows >System32 >WindowsPowerShell.



The Get-Help cmdlet can be expanded with various parameters. For example, if you enter another cmdlet after Get-Help and then add the Examples parameter, you will have examples on how that cmdlet is used. That way, the Get-Help command can help us decide which cmdlet is the best at every occasion.

```
PS C:\Users\PeterKalmström> Get-Help New-PnPSite -Examples|
```

## 13.1.2 Steps

In this exercise, we will build a foreach loop that creates root sites of the Communication type in new site collections. Each department in the Departments array will have such a new site collection.

We will use the cmdlet, New-PnPSite for the creation, and the site type is CommunicationSite.

The URL for each Communication site will end with the department name, and besides actually creating the sites, the foreach will also create a progress message.

1. Create a foreach loop with the statement \$Dep in \$Departments.
2. After the start curly bracket, enter the new variable SiteURL.

3. Add the SiteURL value as a string. It should be the URL to the Sites library in the SharePoint tenant that you want to use + the variable for the current department.

```
foreach($Dep in $Departments){
 $SiteURL = "https://kdemo.sharepoint.com/sites/" + $Dep
```

4. Add a yellow progress message for the site URL creation.

```
Write-Host ($SiteURL + " is being created ...") -ForegroundColor Yellow
```

5. Enter the cmdlet, New-PnPSite.
6. Add the Type parameter and select the value CommunicationSite.
7. Add the Title parameter and give it the value of the current department.
8. Add the Url parameter and give it the value of the SiteURL variable.
9. Finish the foreach loop, so that it looks like this:

```
foreach($Dep in $Departments){
 $SiteURL = "https://kdemo.sharepoint.com/sites/" + $Dep

 Write-Host ($SiteURL + " is being created ...") -ForegroundColor Yellow

 New-PnPSite -Type CommunicationSite -Title $Dep -Url $SiteURL
}
```

10. Run the foreach loop and check that the progress messages are shown for each site URL and that the sites have been created in SharePoint.

As we have created modern sites, you can see the new site collection in the modern SharePoint Admin center. Note that it might take a minute or two before changes are displayed there.

```

PS C:\Users\PeterKalmström> foreach($Dep in $Departments){
 $SiteURL = "https://m365x135719.sharepoint.com/sites/" + $Dep
 Write-Host ($SiteURL + " is being created ...") -ForegroundColor Yellow
 New-PnPSite -Type Communicationsite -Title $Dep -Url $SiteURL
}
https://m365x135719.sharepoint.com/sites/HR is being created ...
https://m365x135719.sharepoint.com/sites/HR
https://m365x135719.sharepoint.com/sites/Marketing is being created ...
https://m365x135719.sharepoint.com/sites/Marketing
https://m365x135719.sharepoint.com/sites/IT is being created ...
https://m365x135719.sharepoint.com/sites/IT
https://m365x135719.sharepoint.com/sites/Sales is being created ...

```

## 13.2 REMOVE SITES

To remove the Communication sites again, we will use the cmdlet `Remove-PnpTenantSite`. The steps below are the same for all kinds of site collections, as only the `Url` is needed.

### 13.2.1 Steps

We will add two other parameters to the mandatory `Url`, to force the deletion and skip adding the removed sites the recycle bin.

1. Copy the create site foreach loop we built above and paste it below the original.
2. Change the progress message so that it becomes red and informs that the site URL is being removed.

```
Write-Host ($SiteURL + " is being removed ...") -ForegroundColor Red
```

3. Remove the row before the end curly bracket and instead enter the cmdlet `Remove-PnpTenantSite`.
4. Add the `Url` parameter and the value of the `SiteURL` variable.
5. Add the parameters `Force` and `SkipRecycleBin`.

```
foreach($Dep in $Departments){
```

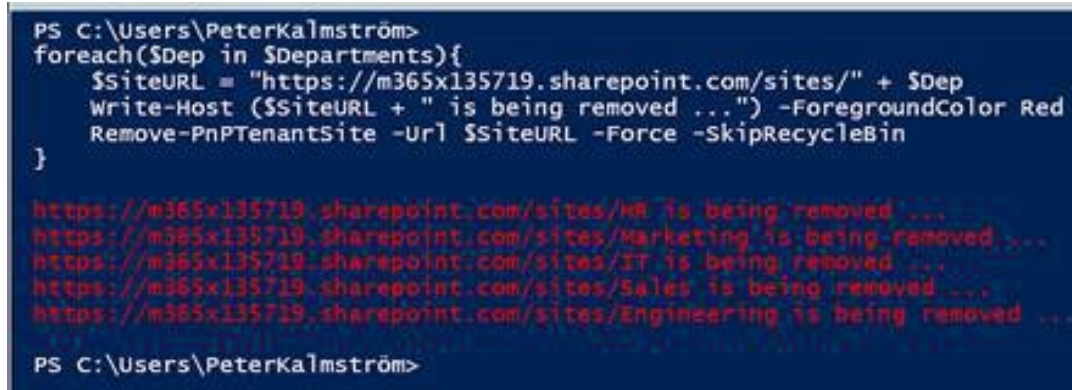
```
 $SiteURL = "https://kdemo.sharepoint.com/sites/" + $Dep
```

```
Write-Host ($SiteURL + " is being removed ...") -ForegroundColor Red
```

```
Remove-PnPTenantSite -Url $SiteURL -Force -SkipRecycleBin
```

```
}
```

6. Run the foreach loop and check that the progress messages are displayed and that the site collections are removed from SharePoint and not left in the recycle bin.



```
PS C:\Users\PeterKalmström>
foreach($Dep in $Departments){
 $SiteURL = "https://m365x135719.sharepoint.com/sites/" + $Dep
 Write-Host ($SiteURL + " is being removed ...") -ForegroundColor Red
 Remove-PnPTenantSite -Url $SiteURL -Force -SkipRecycleBin
}

https://m365x135719.sharepoint.com/sites/HR is being removed ...
https://m365x135719.sharepoint.com/sites/Marketing is being removed ...
https://m365x135719.sharepoint.com/sites/IT is being removed ...
https://m365x135719.sharepoint.com/sites/Sales is being removed ...
https://m365x135719.sharepoint.com/sites/Engineering is being removed ...

PS C:\Users\PeterKalmström>
```

Demos:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/090-Modern-Site-Collections-PowerShell-SharePoint.htm>

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/100-Remove-Site-Collections-PowerShell-SharePoint.htm>

## 13.3 SUMMARY

As you hopefully have seen, it is not very complicated to create new site collections with a PowerShell script – or to remove them! Here we have done it with a function and a foreach loop to create multiple site collections. In later chapters, you will learn how to expand a site collection with subsites, apps, pages and more.

But that will come at the end of the book. Now we will first import data to SharePoint – first Excel items and then files. With PowerShell you can customize such data transfers in a much better way than if you do it directly from SharePoint.

## ***14 IMPORT DATA FROM EXCEL - ONE COLUMN***

Both SharePoint and Excel have functions for transferring data from Excel to SharePoint, and I have described those in my book *SharePoint Online from Scratch*. However, with methods that go directly from Excel to SharePoint, you cannot control exactly what is being transferred and how it is done.

By using PowerShell, you can define which Excel data that should be exported to SharePoint and in which column it should be added, so even if it takes longer to create the scripts I describe below, it might be worth the effort.

Another reason for my inclusion of data transfer from Excel to SharePoint with PowerShell in this book, is that I can show several good techniques that I hope you will find useful in other contexts too.

I will describe two methods for import of data from Excel to SharePoint via PowerShell. The first method can only be used for a list with one column. In the next chapter, we will use an Excel list with multiple columns.

In this chapter, we have an Excel list with just one column, Country. The SharePoint list, to where the data will be added, is a generic list called Countries.

The method we will use, reminds of the one I introduced in chapter 8, where we use multiple Create-MyList calls. We will create a number of calls to a function here too, one for each country value in Excel. We will create those calls in Excel and paste them into PowerShell ISE.

The new factor in this chapter, is that we need to tell PowerShell in which SharePoint column the data should be placed. We cannot just use only the list name.

### **14.1 THEORY**

In this chapter we will use a hashtable with key-value pairs.

A hashtable is a data structure that stores a collection of data. The data is

contained within curly brackets, and the hashtable start is indicated by the @ sign immediately before the start curly bracket.

We have earlier used another structure for data collections that reminds of the hashtable: the array. In an array, the elements are entered in sequence and separated by a delimiter. That is not the case with a hashtable.

Hashtables are most often used for one or more so called key-value, or attribute-value, pairs. These are convenient to use when you have an object – the key – and a specific value that you want to link to that key.

In its most simple form, a key-value pair can look like this:

```
@{'Title'='Argentina'}
```

Here the internal name for a SharePoint column is used with a hardcoded value. We must use the internal name as the key. It is not possible to use the UI column name.

The parameter Values often has a hashtable as its value. When we use the cmdlet Add-PnPListItem and give the list name and the hashtable, we can put Argentina into the SharePoint column that has the internal name Title.

```
Add-PnPListItem -List "Countries" -Values @{'Title'='Argentina'}
```

However, we want all the countries in the Excel list to be added to SharePoint, not just Argentina. Therefore, we will create a variable for the name of the country and use it as the value component in a hashtable. We will add the command in a function that we will call once for each country.

```
function Add-Country([string] $CountryName){
 Add-PnPListItem -List "Countries" -Values @{'Title'=$CountryName}
}
```

A hashtable with key value pairs is also called a dictionary or an associative array.

It is important to remember that the value of a key-value pair is in fact an object and can be a much more complex object than just a string or an integer. It could for example be an instance of a class that has its own methods and properties.

However, a hash table does not have to contain key-value pairs. It is considered as one object in PowerShell, and therefore it can have properties and methods. In chapter 29 we will contain the properties of a video in a hashtable.

## 14.2 STEPS

We will first create a new SharePoint Countries list for the Excel data.

Then we will build a function as described above and use Excel to get the correct function calls for all the countries.

1. Call the Create-MyApp function from chapter 12 to create a generic SharePoint list with the name Countries.

```
Create-MyApp -ListName Countries -AppType GenericList
```

2. Create a function with the name Add-Country and the input parameter CountryName, which should be a string.

```
function Add-Country([string] $CountryName){
```

3. Enter the cmdlet Add-PnPListItem, to add list items to the SharePoint Countries list.

4. Add the List parameter and its value, Countries.

5. Add the Values parameter. As its value, add a hashtable with a key-value pair where the key is the internal name of the SharePoint list column, Title, and the value is the function's CountryName variable.

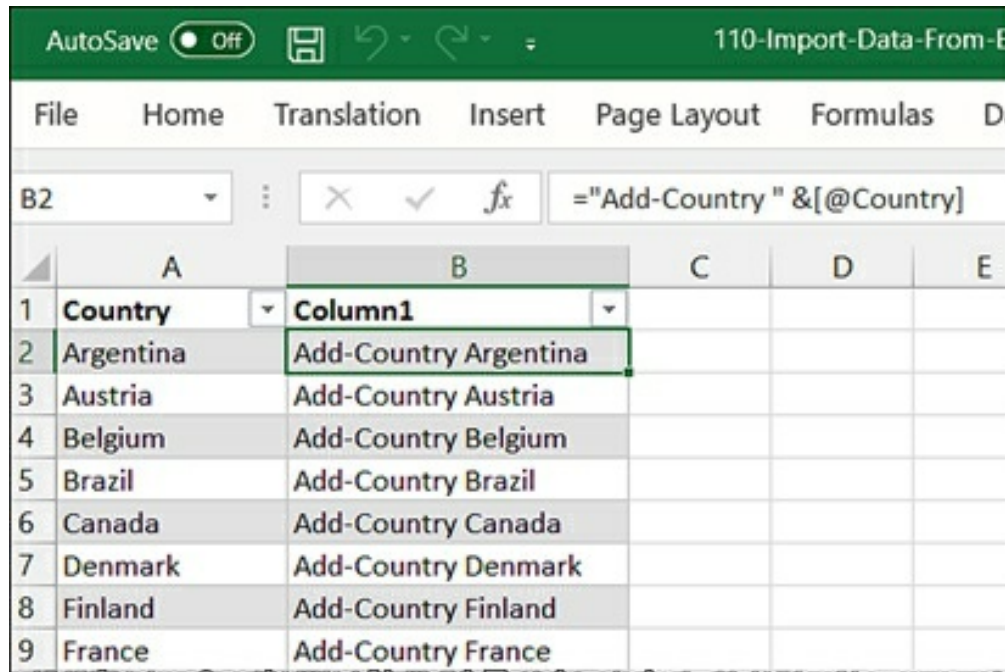
```
function Add-Country([string] $CountryName){
 Add-PnPListItem -List "Countries" -Values @{"Title"=$CountryName}
}
```

6. Initialize the function.

7. Now we can create calls to the function for all the country names. In Excel, enter ="Add-Country " & in the first cell to the right of a country name, B2 in the image below.



8. Select the Argentina cell. This will add [@Country] to the formula.
9. Press enter, and all cells will be filled out with Add-Country and the correct country name.



10. Copy the content of the column – B – and paste it into PowerShell ISE. Now we have function calls for all the countries.

Add-Country Argentina  
Add-Country Austria  
Add-Country Belgium  
Add-Country Brazil  
Add-Country Canada  
Add-Country Denmark  
Add-Country Finland  
Add-Country France

11. Run all the pasted commands in one go and check that the items are being created in the SharePoint list.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/110-Import-Data-From-Excel-PowerShell-SharePoint.htm> – first part

## **14.3 SUMMARY**

In this chapter you have seen how we can use a hashtable with key value pairs in a function that adds data to a specific column in a SharePoint list.

In the next chapter, we will use a foreach loop with such hashtables to create a more advanced call that adds data from Excel to multiple columns in SharePoint.

## ***15 IMPORT DATA FROM EXCEL - MULTIPLE COLUMNS***

In this exercise, we will import Excel data to a SharePoint Contacts list. The automatically created column names of that app type are not the same as in our Excel file. That is not a problem, because a hashtable with key-value pairs can direct Excel data to the correct SharePoint column even if the column names are not the same in Excel and SharePoint.

When we use key-value pairs with the internal column name as the key and the Excel column name as the value, PowerShell will know where in the SharePoint list each object from Excel should be placed.

### **15.1 THEORY**

Here I will introduce another operator and several new cmdlets and explain how they are used. We will also once again use hashtables with key-value pairs, now in a more advanced context than in the previous chapter.

#### **15.1.1 The += Operator and Key-Value Pairs**

The assignment operator += can either increase the value of a variable by a specified value or append a specified value to the existing value.

In this chapter we use a collection of key-value pairs to link Excel columns to internal column names in a SharePoint list. Therefore, the += operator is here used to append the next key-value pair to the existing hashtable. We must start with an empty hashtable and then append the objects from the subsequent hashtable in the run.

```
$RowValues =@{}
```

```
$RowValues +=@{'Title' = $Row.'Contact last name'}
```

#### **15.1.2 Get-Content and Text Files**

In the previous chapter we just copied the Excel content we needed, but here we will use the cmdlet Get-Content to fetch the data from Excel.

The Get-Content cmdlet works best with text, so we will save the Excel file as text to use it with Get-Content. We select to use the tab delimiter and not comma, because there might be commas in the cell values.

We will use a Text variable for the array of Excel data that the cmdlet Get-Content fetches from the text file.

```
$Text = Get-Content "C:\Users\PeterKalmström\Documents\Customers.txt"
```

### 15.1.3 ConvertFrom-Csv

To get the data from each Excel row in a structured way, we will create a Rows array. The cmdlet for reading from the .txt file is ConvertFrom-Csv, which analyses CSV strings and creates objects from them.

CSV means comma separated value, but this cmdlet can be used even though our text file is tab separated. We just need to give the tab separated value to the Delimiter parameter: a backtick sign together with a t.

```
$Rows = ConvertFrom-Csv -Delimiter "`t" -InputObject $Text
```

The parameter InputObject is the object that PowerShell should be working with, which here is the Text variable that gets its content from the text file, see above.

### 15.1.4 Get-PnPFields

The cmdlet Get PnPFields fetches all columns from a specified SharePoint app, including all columns that are hidden from the user interface. The result is shown in three columns in the console pane: Title, (the column name that is shown in the UI), Internal name and Id.

In this exercise, we will use the cmdlet Get PnPFields to find internal column names. These names will be used as keys in key-value pairs, and the value will be the corresponding Excel heading.

```
PS C:\Users\PeterKalmström> Get-PnPField -List "Customers"
```

| Title                 | InternalName         | Id                                   |
|-----------------------|----------------------|--------------------------------------|
| ----                  | -----                | --                                   |
| ID                    | ID                   | 1d22ea11-1e32-424e-89ab-9fedbadb6ce1 |
| Content Type ID       | ContentTypeId        | 03e45e84-1992-4d42-9116-26f756012634 |
| Content Type          | ContentType          | c042a256-787d-4a6f-8a8a-cf6ab767f12d |
| Last Name             | Title                | fa564e0f-0c70-4ab9-b863-0177e6ddd247 |
| Modified              | Modified             | 28cf69c5-fa48-462a-b5cd-27b6f9d2bd5f |
| Created               | Created              | 8c06beca-0777-48f7-91c7-6da68bc07b69 |
| Created By            | Author               | 1df5e554-ec7e-46a6-901d-d85a3881cb18 |
| Modified By           | Editor               | d31655d1-1d5b-4511-95a1-7a09e9b75bf2 |
| Has Copy Destinations | _HasCopyDestinations | 26d0756c-986a-48a7-af35-bf18ab85ff4a |
| Copy Source           | CopySource           | 6b4a226d-3d88-4a36-808d-a129b452bcef |

Note that PowerShell mostly uses the word "field" for "column". There are no differences, here field and column is the same thing.

## 15.2 STEPS

In this exercise, we will use an Excel list with contacts data. It has columns for first name, last name, e-mail and company. Each of the Excel columns has a heading that tells what kind of data you can find in the cells below.

We will save the Excel file as a text file and use the Get-Content cmdlet to get data from that file into a Text variable. A Rows array will then read the Text variable for each Excel row.

Finally, a foreach loop runs through the Rows array and creates new items in a SharePoint contacts list. The values for each new item are defined in a RowValues variable with a collection of key-value pairs.

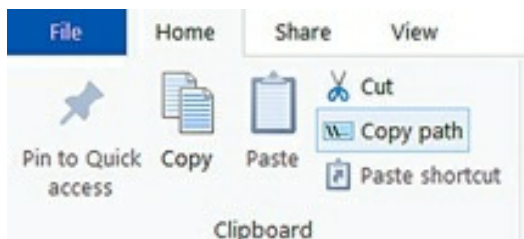
1. Call the Create-MyApp function from chapter 12 to create a SharePoint contacts list with the name Customers.

`Create-MyApp -ListName Customers -AppType Contacts`

2. Save the Excel file as a tab delimited text file with the name Customers.



3. Copy the path to the Excel file from Windows Explorer >Home >Copy path.



4. In PowerShell ISE, enter the variable Text and give it the value of an array that the cmdlet Get-Content fetches from the path you copied.

```
$Text = Get-Content "C:\Users\PeterKalmström\Documents\Customers.txt"
```

5. Run the Text variable and check that it gives you the correct output. It will look like in the image below – confusing, but you should be able to recognize data from the Excel list.

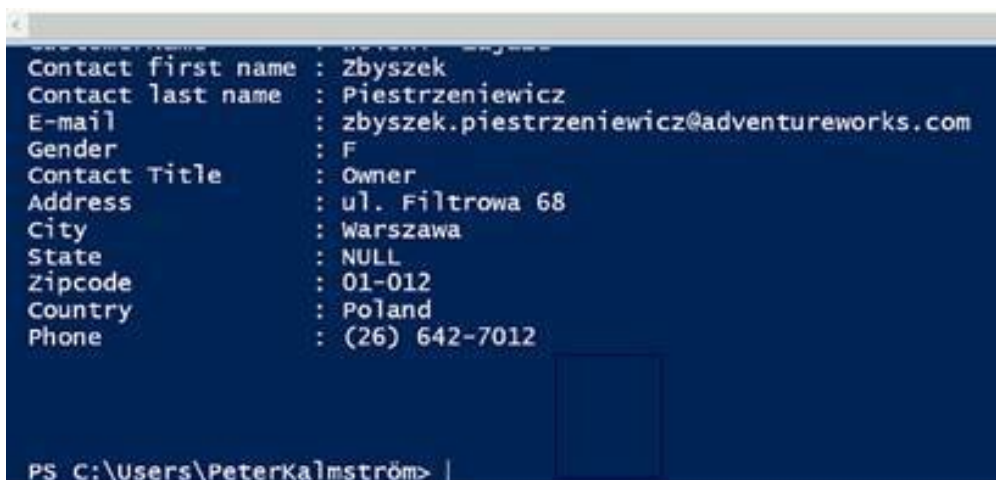


```
VICTE victualles en stock Mary Savoley mary.savoley@adventureworks.com F Sales Agent 2, rue du Commerce Lyon NULL 69006 France
VINET vins et alcools Chevalier Paul Henriot paul.henriot@adventureworks.com M Accounting Manager 59 rue de l'Abbaye Reims NULL 5110
7.15.30
WANDK Ofc Wandende kuh Rita Müller rita.mueller@adventureworks.com F Sales Representative Adenauerallee 900 Stuttgart NULL 7054
861
WARTI wärtian Herkku Pirkko Koskitalo pirkko.koskitalo@adventureworks.com F Accounting Manager Torikatu 38 Oulu NULL 90110 Finland
WELLI wellington Importadora Paula Parente paula.parente@adventureworks.com F Sales Manager Rua do Mercado, 12 Resende SP 08737-363
55-8322
WHITE White Clover Markets Karl Jablonski karl.jablonski@adventureworks.com M Owner 305 - 14th Ave. S, Suite 38 Seattle WA 98128
12
WILMA Wilman Kale Matti Karttunen matti.karttunen@adventureworks.com M Owner/Marketing Assistant Keskuskatu 45 Helsinki NULL 2120
8858
WOLZA wolski Zajazd Zbyszek Piestrzeniewicz zbyszek.piestrzeniewicz@adventureworks.com F Owner ul. Filtrowa 68 Warszawa NULL 01-012
642-7012
PS C:\Users\PeterKalmström>
```

6. To get the data ordered in proper rows for the SharePoint list, create a new variable, Rows, on a new row.
7. Give the Rows variable the value of an array created by the cmdlet ConvertFrom-CSV.
8. Add the Delimiter parameter with the value "`t".
9. Add the InputObject parameter with the value of the Text variable we created in point 4.

```
$Rows = ConvertFrom-Csv -Delimiter "`t" -InputObject $Text
```

10. Check that the Rows variable gives you the correct content by running \$Rows at the command prompt. The output of properties and their values will now look much more structured than in the image above.



```
Contact first name : Zbyszek
Contact last name : Piestrzeniewicz
E-mail : zbyszek.piestrzeniewicz@adventureworks.com
Gender : F
Contact Title : Owner
Address : ul. Filtrowa 68
City : Warszawa
State : NULL
Zipcode : 01-012
Country : Poland
Phone : (26) 642-7012
PS C:\Users\PeterKalmström>
```

11. For further checking and IntelliSense, add `$Row = $Rows [0]` in the script pane and run it. Then run `$Row` at the command prompt. That will give you the data from the first object in the array – which is the first row in the text file. (I explained this in more detail in chapter 9, Array.)
12. Comment out or remove the line `$Row = $Rows [0]`, because we don't want just the first object – we want all of them.
13. Create a foreach loop that runs through the Rows array.
14. Enter the cmdlet `Add-PnPListItem` after the start curly bracket and add the List parameter with the value of the Customers list we created in point 1.
15. Add the Values parameter and give it the value of a RowValues variable (not yet created).

```
foreach($Row in $Rows){
 Add-PnPListItem -List "Customers" -Values $RowValues
}
```

16. Enter a new RowValues variable after the start curly bracket in the foreach loop, before the `Add-PnPListItem` command. (As everything in a script is executed in row order, the definition must be written before the actual execution.)
17. Give the RowValues variable the value of an empty hashtable.

```
$RowValues =@{}
```

18. At the command prompt, enter the cmdlet `Get-PnPField` with the List parameter and the value Customers.
19. Run the `Get-PnPField` command.

```
Get-PnPField -List "Customers"
```

20. Copy the result from the command pane and paste it into a text file, so that you can more easily look up the internal names.

```

Home Phone HomePhone 2a092320-9080-4b91-9083-c0f990009107
Mobile Number CellPhone 2a464df1-44c1-4851-949d-fcd270f0ccf2
Fax Number WorkFax 9d1cacc8-f452-4bc1-a751-050595ad96e1
Address WorkAddress fc2e188e-ba91-48c9-9dd3-16431afddd50
City WorkCity 6ca7bd7f-b490-402e-af1b-2813cf087b1e
State/Province WorkState ceac61d3-dda9-468b-b276-f4a6bb93f14f
ZIP/Postal Code WorkZip 9a631556-3dac-49db-8d2f-fb033b0fdc24
Country/Region WorkCountry 3f3a5c85-9d5a-4663-b925-8b68a678ea3a
Web Page WebPage a71affd2-dcc7-4529-81bc-2fe593154a5f
Notes Comments 9da97a8a-1da5-4a77-98d3-4bc10456e700

```

P5 C:\Users\PeterKalmström>

21. Copy \$RowValues =@{} in the foreach loop and paste it on the row below.
22. Replace the assignment operator = with +=
23. Enter the value of the first SharePoint column, Title, after the start curly bracket. This is the key.
24. Add the value of the current row with the property Contact last name, so that the Title column will show the last name of the contact person.

As you ran the \$Row = \$Rows[0] command above in point 11, you will get IntelliSense when you type the dot after \$Row. This is very helpful and good for avoiding mistakes.

```

foreach($Row in $Rows){
 $RowValues =@{}
 $RowValues +=@{'Title' = $Row.'Contact last name'}
}

```

25. Copy the row \$RowValues +=@{'Title' = \$Row.'Contact last name'} you just wrote and paste it three times on the rows below.
26. Change each of the pasted rows, so that you add the internal SharePoint name for another column as the key and the Row variable with the property of the corresponding Excel column heading as the value.
27. Now the whole foreach loop looks like this:

```

foreach($Row in $Rows){
 $RowValues =@{}
 $RowValues +=@{'Title' = $Row.'Contact last name'}
 $RowValues +=@{'Company' = $Row.CustomerName}
 $RowValues +=@{'FirstName' = $Row.'Contact first name'}
}

```



```
$RowValues +=@{'Email' = $Row.'E-mail'}
Add-PnPListItem -List "Customers" -Values $RowValues
}
```

Notice that the object properties that comes from Excel column headers with spaces have property names surrounded by single quotes.

28. Run the foreach loop and check that the items are being created in the SharePoint list and that the data from Excel is being correctly distributed in the columns.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/110-Import-Data-From-Excel-PowerShell-SharePoint.htm> – second part

## 15.3 SUMMARY

In this chapter, I have explained how you can fetch Excel data with the Get-Content cmdlet and get it into the SharePoint columns you prefer by using key-value pairs.

In the following chapters, we will work with files instead.

# ***16 ADD FILES TO SHAREPOINT – UPLOAD FILES***

SharePoint has several methods to upload files from a local folder to a document library, but if you want to do it automatically, you should use a PowerShell script.

In such a scenario, you can put all files that should be uploaded in a specific folder, and then they will be uploaded automatically to the SharePoint library of your choice.

In this exercise, we will first write a script that copies all files in a folder to a SharePoint library.

In the next chapter, we will expand the script to have all copied files moved to a subfolder.

Finally, we will set the Windows Scheduler to run the script every 10<sup>th</sup> minute.

## **16.1 THEORY**

Here I will introduce the two cmdlets we will use in the exercise. I will also explain how we can get IntelliSense for a foreach loop that has not yet been run.

### **16.1.1 Get-ChildItem**

In this exercise, we will create a Files variable and set an array created by a Get-ChildItem command to its value. It will then be used in another command to send files to SharePoint.

```
$Files = Get-ChildItem "C:\Users\PeterKalmström\Documents\ToImport"
```

The Get-ChildItem cmdlet gets items from a specified container, in this case a Windows Explorer folder. Here we will only use the parameter Path and its value, the path to the folder.

The Path value can however be expanded to only get files of a certain format.

```
Get-ChildItem -Path "C:\Users\PeterKalmström\Products*.pdf"
```

Get-ChildItem can also be expanded with multiple parameters. The Exclude parameter with the value string \*.docx would exclude all the Word files, for example.

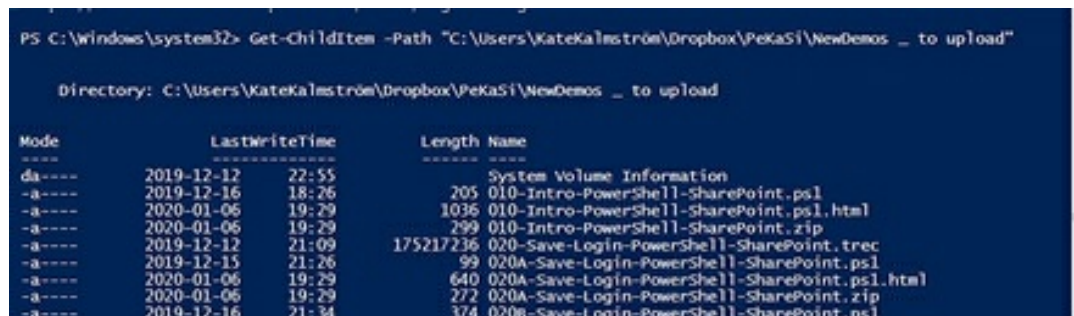
```
Get-ChildItem -Path "C:\Users\PeterKalmström\Products" -Exclude "*.docx"
```

Another Get-ChildItem parameter is Recurse, which lets the script get items in all subfolders, and Depth, which limits the number of levels to recurse. The parameter File limits the output to files, excluding folders.

We will not use any extra parameters in this exercise, but Recurse and File will be used in chapter 19.

By default, the cmdlet Get-ChildItem only fetches non-hidden items. Use the Force parameter to include hidden items in the results.

The result in the console pane is given with Mode, LastWriteTime, Length (=size) and Name. The modes are l (link), d (directory), a (archive), r (read-only), h (hidden) and s (system).



```
PS C:\Windows\system32> Get-ChildItem -Path "C:\Users\KateKalmström\Dropbox\PeKaSi\NewDemos _ to upload"

Directory: C:\Users\KateKalmström\Dropbox\PeKaSi\NewDemos _ to upload

Mode LastWriteTime Length Name
---- -
da----- 2019-12-12 22:55 System Volume Information
-a----- 2019-12-16 18:26 205 010-Intro-PowerShell-SharePoint.ps1
-a----- 2020-01-06 19:29 1036 010-Intro-PowerShell-SharePoint.html
-a----- 2020-01-06 19:29 299 010-Intro-PowerShell-SharePoint.zip
-a----- 2019-12-12 21:09 175217236 020-Save-Login-PowerShell-SharePoint.trec
-a----- 2019-12-15 21:26 99 020A-Save-Login-PowerShell-SharePoint.ps1
-a----- 2020-01-06 19:29 640 020A-Save-Login-PowerShell-SharePoint.ps1.html
-a----- 2020-01-06 19:29 272 020A-Save-Login-PowerShell-SharePoint.zip
-a----- 2019-12-16 21:34 374 020A-Save-Login-PowerShell-SharePoint.ps1
```

If you only want to see the names, add the parameter Names without any value to the command.

## 16.1.2 Add-PnPFile

When we have added the files from the Windows Explorer folder to an array with the Get-ChildItems cmdlet, we can add them to the SharePoint library with the cmdlet Add-PnPFile.

The Add-PnPFile cmdlet requires the parameters Folder and Path. The Folder value should be the name of the SharePoint library. The Path is the path to each object in the local folder.

To find the path for each object, we will create a foreach loop that runs

through the array of files. We can use the current File element in the array to get the path for each file in the folder.

```
foreach($File in $Files){
 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
}
```

The Add-PnPFile cmdlet cannot upload subfolders. In the next chapter, when we will add a subfolder and move the copied files to it, we need to tell PowerShell to ignore subfolders. Otherwise the script will give an error message for the subfolder.

### 16.1.3 Foreach Loops and IntelliSense

As we saw in chapter 9, PowerShell creates the current element variable automatically when a foreach loop is run through an array. That means that no value is set until we run the foreach loop, and when there is no value there is no IntelliSense.

To get IntelliSense for the array elements in PowerShell ISE during the creation of the foreach loop, we must run through the array for one of the elements inside the foreach loop.

I usually add this line of code to the loop:

```
$Dep = $Departments[0]
```

This means that I assign the Dep variable to the first item in the Departments array. "The current department that is run, is equal to the first department in the Departments array". When you have run that row, comment out or remove it. Otherwise, the foreach loop will just do the same thing over and over again against the first element in the array, which is not what we want.

## 16.2 STEPS

We will upload files from a Windows Explorer folder called To-Import to the SharePoint default document library, Shared Documents.

1. Copy the path to the To-Import folder in Windows Explorer.
2. Create a Files variable with a value that contains an array of all the objects

in the folder. Get these files with the Get-ChildItem cmdlet and the path you copied in step 1.

```
$Files = Get-ChildItem "C:\Users\PeterKalmström\Documents\ToImport"
```

3. Initialize the variable.
4. Enter \$Files at the command prompt to check that the files in the folder are displayed as they should in the console pane.
5. Start building a foreach loop that runs through all the files.
6. To get IntelliSense suggestions for the file object, add the code \$File = \$Files[0] after the start curly bracket and run it.
7. Enter the Add-PnPFile cmdlet after the start curly bracket, to add the files to SharePoint.
8. Add the parameter Folder and add a string with the SharePoint document library name as its value.
9. Add the parameter Path and the value \$File.
10. When you enter a dot after the value, you will get property suggestions. Select Fullname to include the file extension.
11. Comment out or remove the code you added in point 5, because we want the foreach loop to run through all the objects and not only the first one.

```
foreach($File in $Files){
 # $File = $Files[0]

 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
}
```

12. Run the foreach loop and check that all the files have been uploaded to the SharePoint library.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/120-Upload-Files-PowerShell-SharePoint.htm>

## 16.3 SUMMARY

In this exercise we have created a foreach loop that runs through all the files in a Windows Explorer folder and uploads them to a SharePoint library.

We used the cmdlet `Get-ChildItem` to get the files into an array that the foreach loop could run through.

The foreach loop then used a command with the cmdlet `Add-PnPFile` to fetch the files from the array and add them to SharePoint.

In the next chapter we will expand the foreach loop to move the files to a subfolder when they have been uploaded.

## ***17 ADD FILES TO SHAREPOINT – MOVE UPLOADED FILES***

In the previous chapter, we created a script that copies files from a local folder to a SharePoint document library. Now, we will enhance that script and let it move the copied files to a subfolder.

This is of course necessary at automatic uploading. You don't want the already uploaded files to be uploaded again and maybe overwrite later versions that have been changed in SharePoint.

The subfolder should not be copied to SharePoint, so we will have to tell PowerShell to not include folders in the upload. We will do that by adding an If statement to the foreach loop that we created in the previous chapter.

Another reason for the If statement is that the Add-PnPFile cmdlet we use for the upload to SharePoint in this and the previous exercise cannot upload subfolders. Therefore, the script will fail and give an error without the If statement.

### **17.1 THEORY**

In this theory section, I will describe error handling and how to make the script ignore the subfolder when we use the Add-PnPFile cmdlet.

#### **17.1.1 Error handling**

If it is possible, PowerShell continues running after reporting an error in the console pane. This default behavior does not suit me, because when there is an error, I want to stop the script and fix the error before I continue running it.

To change the default error handling, add two lines of script near the top of the script. I usually put them right after the URL variable for the SharePoint connection.

```
$ErrorActionPreference = "Stop"
```

```
$PSDefaultParameterValues['*:ErrorAction']='Stop'
```

You must run these two lines to make the error handling settings for the current session.

### 17.1.2 Find Folder

The Get-Childitem cmdlet that we are using to get the files from Windows Explorer returns both files and folders. Folders – but not files – have a property called PSIsContainer, and we can use it to decide if the current object is a folder or not.

```
if($File.PSIsContainer -ne $true){
```

### 17.1.3 Comparison Operators

We need to find out if the current object that the foreach loop runs through is a folder or not. This is coded as -eq for equal and -ne for not equal. We will use an If statement with the -ne operator in this exercise, *see* the line of code above.

PowerShell has many such comparison operators. Here are the most common ones:

- -eq equal
- -ne not equal
- -ge greater than or equal
- -gt greater than
- -lt less than
- -le less than or equal

## 17.2 STEPS

We will move the files that have been copied to SharePoint to a subfolder within the To-Import folder, so the first step is to create such a subfolder. We will call that subfolder Imported.

After that, we can start working in PowerShell ISE and continue adding code to the foreach loop we created in chapter 16.



```
foreach($File in $Files){
 #File = $Files[0]

 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
}

```

We will add code that will exclude the subfolder from the upload and move the copied files to the subfolder.

Start on a new row before the end curly bracket of the foreach loop.

1. Enter the cmdlet Move-Item.
2. Add the Path parameter and its value, the full name of the current file.
3. Add the parameter Destination and its value: the URL to the subfolder in Windows Explorer to where the files should be moved after being copied to SharePoint.

```
Move-Item -Path $File.FullName -Destination
"C:\Users\PeterKalmström\Documents\ToImport\Imported"
```

4. Enter an If statement after the foreach loop's start curly bracket: if it is true that the current file is not equal to an object with the PSIsContainer property.
5. Add another start curly bracket after the If statement end parenthesis.

```
if($File.PSIsContainer -ne $true){
```

6. Add another end curly bracket.

7. Now the whole loop looks like this:

```
8. foreach($File in $Files){
 if($File.PSIsContainer -ne $true){
 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
 Move-Item -Path $File.FullName -Destination
 "C:\Users\PeterKalmström\Documents\ToImport\Imported"
```

```
}
}
```

9. Run the foreach loop.
10. Check that all the files in the To-Import folder have been copied to SharePoint and then moved to the Imported subfolder.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/130-Upload-Files-2-PowerShell-SharePoint.htm>

## 17.3 SUMMARY

To automate the upload of files from Windows Explorer, we need to move the copied files into a subfolder. In this chapter, we have seen how that can be done with the cmdlet Move-Item.

We have also excluded the subfolder from the upload with an If statement that checks if the current file has the property PSIsContainer.

In the next chapter, I will explain how to schedule automatic running of the script we have created.

# 18 AUTOMATE A POWERSHELL TASK

To automate the running of PowerShell scripts, you can use Windows Task Scheduler. In this chapter, we will use it for the script we created in the two previous chapters, that copies files to SharePoint and then moves them to a subfolder.

## 18.1 THEORY

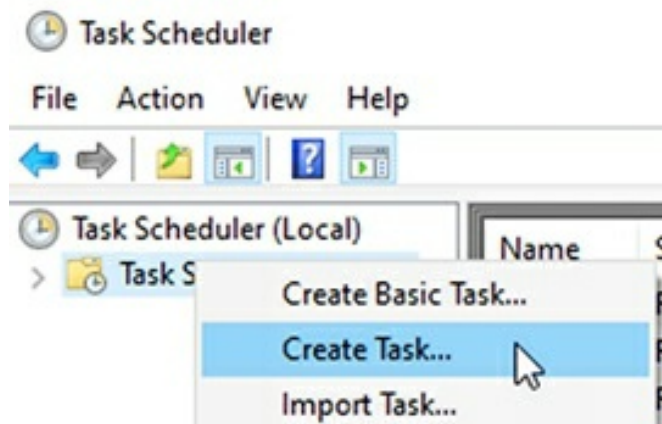
The Task Scheduler is a tool that is included in Windows. It lets you schedule the running of scripts to pre-defined times or after specified time intervals.

The SharePoint credentials must be stored in Windows Credential Manager, *refer to chapter 4*, when you want the Task Scheduler to run PowerShell scripts that connect to SharePoint.

## 18.2 STEPS

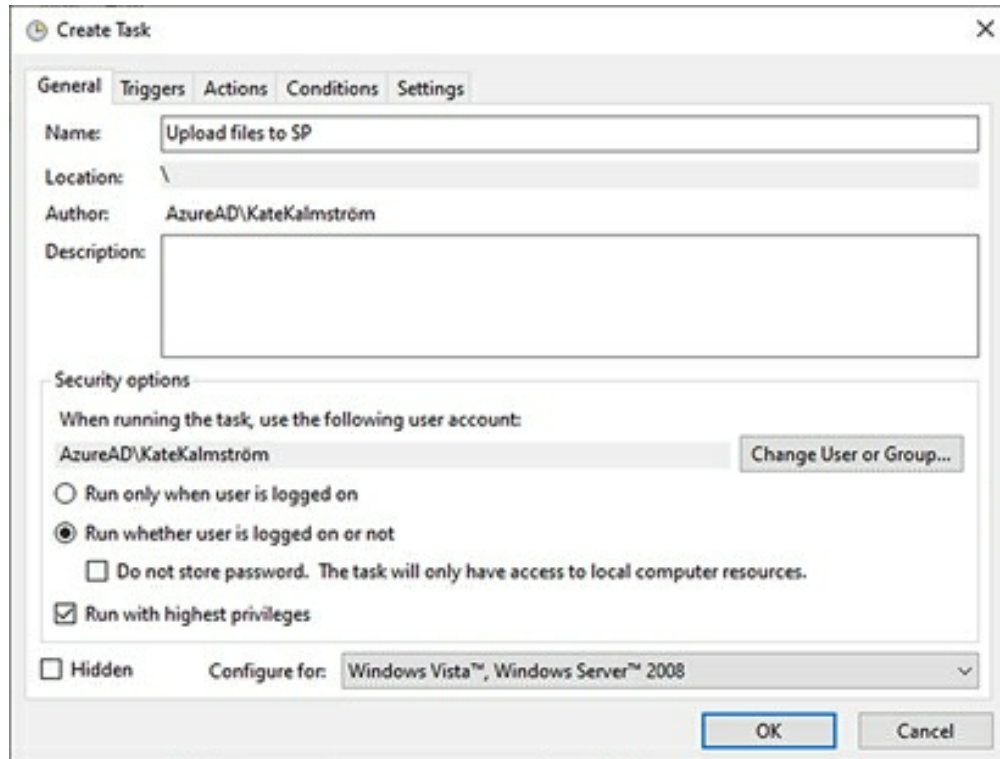
Here we will set the Task Scheduler to run the script every 10<sup>th</sup> minute. If you are connected to Azure AD, it is necessary to use the option Run with highest privileges.

1. In the Task Scheduler, right-click on Task Scheduler Library and select Create Task...



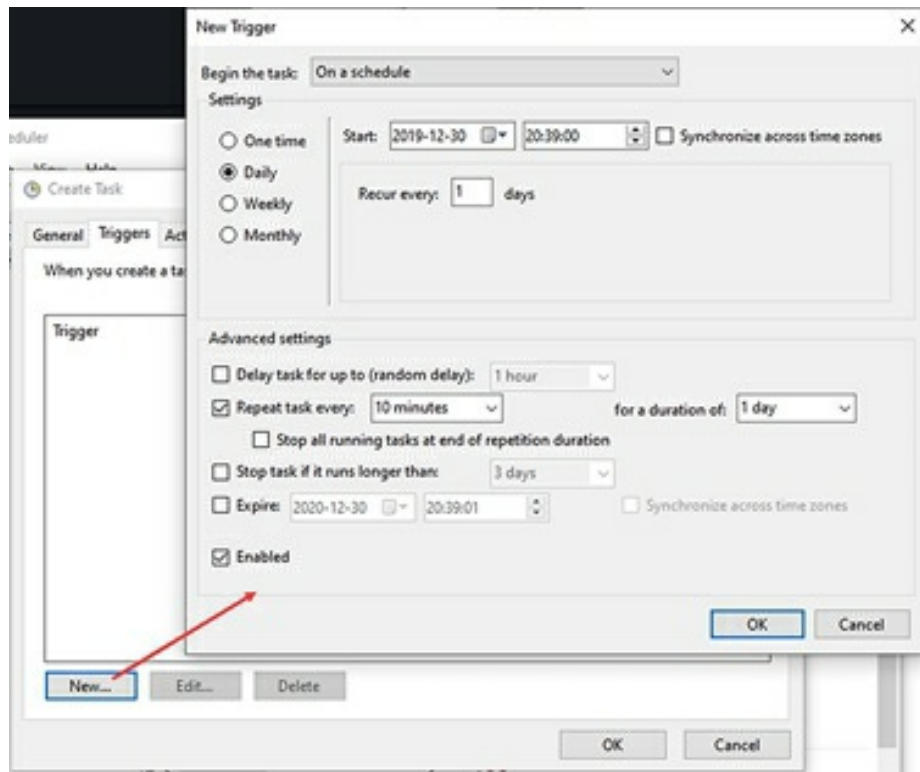
2. Give the new task a name.
3. Set the task to run whether users are logged on or not

4. (Set the task to be run with highest privileges.)

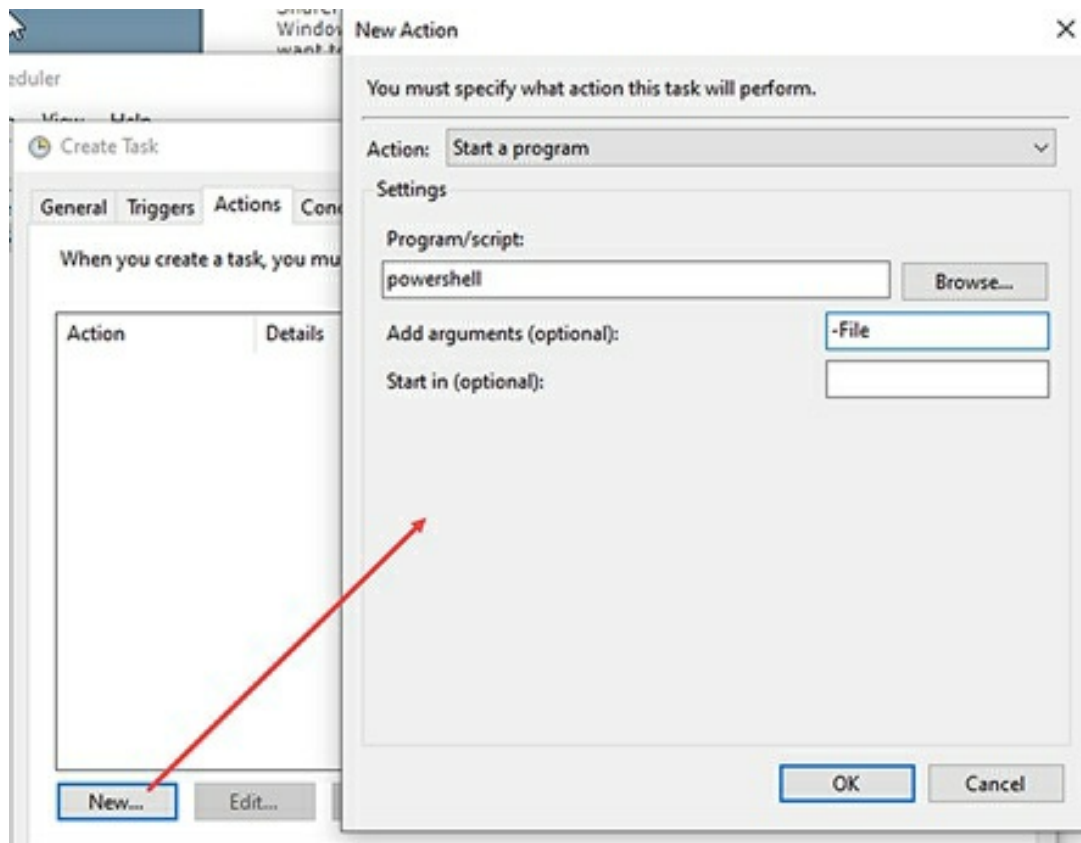


5. Open the Triggers tab and click on New.

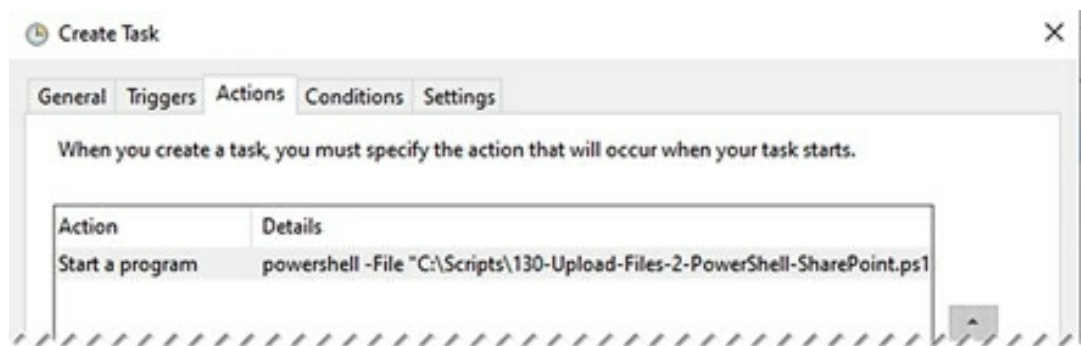
6. Set the task to run daily, every 10<sup>th</sup> minute.



7. Open the Action tab and click on New.
8. Keep the default Action and set the program to powershell (lower case is recommended here).
9. At Argument (which is the same as parameter), add -File and the path to the PowerShell script. Quotation marks will be added automatically around the path.



10. Click OK when you have added the path.



11. Click OK again to save the task.

12. Add some files to the folder and make sure they are uploaded to SharePoint and then moved to the subfolder within 10 minutes.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/140-Scheduled-Tasks-PowerShell-SharePoint.htm>

## **18.3 SUMMARY**

This chapter had no new PowerShell theory. Instead, I have shown how you can set Windows Task Scheduler to run scripts. I used the file upload script as an example, but you can of course automate the running of other scripts in the same way.

In the following three chapters, we will continue with file upload, but now I imagine a folder with many files that will be uploaded once to SharePoint for continued storing and editing. In such a case, it is important to include metadata in the upload.

## ***19 UPLOAD FILES WITH METADATA – LIBRARY***

In the previous chapters, we created a script that uploads files from a local folder to a SharePoint document library and then moves them to a subfolder.

Now we will continue with the same theme, because there is more to think about. I have created a series of Tips articles on things to consider when you upload files to SharePoint: <https://www.kalmstrom.com/PowerShell/Move-Files-Into-SP-Intro.htm>.

One of the things that requires some consideration is that metadata cannot be kept when you use one of the standard SharePoint upload methods. The file names are kept, but you cannot make use of them in SharePoint for filtering and grouping, and other metadata is lost completely. That problem can be solved with PowerShell and a bit of brainwork.

Here we will have a look at how metadata from folders and file names can be retained in a useful way when files are uploaded to SharePoint with a PowerShell script.

I have divided the process into three parts:

- This chapter: Create the SharePoint document library with the metadata columns we need.
- Next chapter: Add each type of metadata to an object that fetches its property from a class. The value of each object is decided by the place where we will fetch the metadata value for each file.

Chapter 21: Upload the files with their metadata and set the correct library column values.


### **19.1 PRERESIQUITES**

I will take a folder with CV files as an example for this exercise. It is called CVs and has three subfolders with the names Hired, Maybe and No.

All files in these three subfolders are named in the same pattern, with the



name of the applicant and the department where they applied for a position:  
Last name, First name – Department.

 Smith, John - Sales

When we copy the files to a SharePoint library, we want to keep the metadata given by the file name and the subfolder placement and structure it in a way that makes it useful in SharePoint.

The metadata will be placed in four different columns: Decision (from the subfolder) and First name, Last name and Department (from the file name). We will create this SharePoint library in the next chapter.

If you want to try these exercises, you should create a similar folder with three subfolders and a few files in each subfolder. The files should be named in the way I have described above.

## 19.2 THEORY

You already know how to add apps to SharePoint via PowerShell, but here we will also create commands for SharePoint columns.

Remember that the columns are called fields in the SharePoint API and that all apps are called lists.

SharePoint app columns are created with the cmdlet `Add-PnPField`. In this exercise, we will add four parameters and their values to this cmdlet:

- `List` - the name of the app where the column should be added.
- `DisplayName` – the GUI name of the column.
- `InternalName` – the internal name value is not always necessary to specify when you create SharePoint columns with PowerShell. However, if you want a PowerShell script to do something with a column (for example add metadata to them) you must use the internal SharePoint name in that script. Therefore, it is a good habit to create internal names that are consistent and easy to remember.
- `Type` – the column type.

If the Type value is Choice or Multichoice, you must add the parameter Choices. The value of the Choices parameter should be the choice options separated with comma.

- **AddToDefaultView.** A parameter without value that adds the column to the default view.

## 19.3 STEPS

We will start with creating a SharePoint document library called CVs, using the New-PnPList cmdlet I introduced already in chapter 5. After that, we will create commands for each library column.

1. Enter the cmdlet New-PnPList.
2. Add the parameters Title and Template, with the values CVs and DocumentLibrary.  

```
New-PnPList -Title "CVs" -Template DocumentLibrary
```
3. Run the command and check that the document library has been created in SharePoint.
4. Enter the cmdlet Add-PnPField.
5. Add the parameter List and the value CVs
6. Add the DisplayName parameter and the value Decision.
7. Add the parameter InternalName and the value CVDecision.
8. Add the parameter Type and the value Choice.
9. Add the Choices parameter and the values of the strings Yes, No and Maybe.
10. Add the parameter AddToDefaultView.

```
Add-PnPField -List "CVs" -DisplayName "Decision" -InternalName "CVDecision" -Type Choice
-Choices "Yes","No","Maybe" -AddToDefaultView
```

11. Run the command and check that the Decision column has been created

in the CVs library.

12. Copy the Add-PnPField command and paste it on three rows below the original.
13. Change the parameters and values so that they apply to the first name, last name and region. All these should be text type columns.

```
Add-PnPField -List "CVs" -DisplayName "First Name" -InternalName "CVFirstName" -Type Text -AddToDefaultView
```

```
Add-PnPField -List "CVs" -DisplayName "Last Name" -InternalName "CVLastName" -Type Text -AddToDefaultView
```

```
Add-PnPField -List "CVs" -DisplayName "Department" -InternalName "CVDepartment" -Type Text -AddToDefaultView
```

14. Run the three new Add-PnPField commands and check in SharePoint that the additional columns have been created.

CVs



Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/150-Upload-Files-With-Metadata-PowerShell-SharePoint.htm>

## 19.4 SUMMARY

Now we have created a new SharePoint library for the CV files we want to upload, and we have used the Add-PnPField cmdlet to create four columns for metadata.

I hope you found this chapter rather easy to follow and are prepared for a more complicated task! You will find that in the next chapter, where we will structure the file metadata we want to keep. For that, we will create a class and split the file names into three parts.

## ***20 UPLOAD FILES WITH METADATA – STRUCTURE***

In the previous chapter, we created a SharePoint document library with four custom columns. Now it is time to structure the metadata we can find in the files we want to upload, so that the metadata values for each file can be set in the correct way in the SharePoint columns.

### **20.1 THEORY**

In this exercise, we will create a variable with an array of all the files in the Windows Explorer folder. A foreach loop will run through the array and find the correct metadata for each column in the SharePoint library.

The creation is done in several steps, and we will check the script continuously while we are creating it. Therefore, we will add a Write-Host cmdlet that will let us check in the console pane that everything is structured in the right way.

`Write-Host (ConvertTo-Csv $CVObj)`

To follow how the script is built, you also need to understand some concepts that we have not used earlier in this book.

#### **20.1.1 Split Operator**

The values for the SharePoint Decision column can be taken from the subfolder names, but the file names must be split into three parts for three different columns: Last Name, First Name and Department. For this we will use the split operator, that splits one or more strings into substrings.

The code below, splits the file name without extension in two parts at a hyphen surrounded by spaces.

`$CVFile.BaseName -split ' - '`

#### **20.1.2 Class**

A class is a kind of model or blueprint that specifies the properties and/or

methods of something. We will not go into class methods in this book, but we will use class properties in this exercise and two later ones.

From PowerShell 5.0 you can create classes directly from within PowerShell ISE in the same way as you create functions.

Class properties can be variables or hardcoded data.

When we call a class, we create an instance of it. We can also say that we instantiate the class. The instance, or object, has access to all the properties that are defined in the class.

There are many built-in classes in PowerShell, and cmdlets often return an instance of a class. I show a very simple example on working with classes in this Tips article:

<https://www.kalmstrom.com/Tips/PowerShell/PowerShell-Json-Object.htm>

### 20.1.2.1 Create a Class

Start a new class with `Class` and the name you want to use when you call the class. Make the name unique, otherwise PowerShell will use the existing class. Then add the class properties within curly brackets.

In this exercise, we are using a class named `CV`, and the properties are the four metadata variables that we have defined earlier: the three parts of the file name and the subfolder that informs about the decision made for each applicant.

```
Class CV{
 $FirstName

 $LastName

 $Department

 $Decision
}
```

When you create a class and give it a name, we say that you declare the class. (Just like you declare a variable when you give it a name.)

As we have seen earlier, variables that work as inputs to a function act as parameters. Variables inside a class definition instead act as properties, but

just like the parameters these properties can have values.

Classes must be initialized before they can be used, just like variables and functions.

### 20.1.2.2 Create an Instance of a Class

From PowerShell 5.0 we can call a class with the cmdlet `New-Object` and the class name. In this exercise, we will use the variable `CVObj` for the new instance that is created when we call the `CV` class.

```
$CVObj = New-Object CV
```

When you have initialized this variable, the properties of the class can be selected if you add a dot after the variable name.



Now we can give a value for each of the class properties. The values can be hard-coded, but here the property value for the current object will be a variable that tells where we can fetch the property value.

```
$CVObj.Decision = $CVFile.Directory.Name
```

## 20.2 STEPS

In this exercise we will create a long script, so I will divide it into several parts to make it easier to follow.

### 20.2.1 Create a CV Class

We will start by creating the `CV` class I described in the theory section above.

1. Create a new class, `CV`, for the four kinds of file metadata that we want to add to SharePoint.

```
Class CV{
 $FirstName
```

```
$LastName
```

```
$Department
```

```
$Decision
```

```
}
```

2. Run the class, to initialize it.

## 20.2.2 Create an AllCVFiles Array

Now we will create an array that contains all the files in the CVs folder.

1. Create an AllCVFiles variable that will contain an array of all the files in the Windows Explorer folder.
2. Set the value of the AllCVFiles variable to an array created by the Get-ChildItem cmdlet and the parameter Path with value of a string with the path to the CVs folder.
3. Add the parameter Recurse, so that the subfolders are included.
4. Add the parameter File, to only include files.

```
$AllCVFiles = Get-ChildItem -Path "C:\Users\PeterKalmström\Documents\CVs" -Recurse -File
```

5. Run the entire AllCVFiles row to initialize the variable. Check that the files from the CVs folder are displayed in the console pane.

## 20.2.3 Create a Foreach Loop for the Array

We will use the class and the array in a foreach loop. This loop will:

- Create instances of the CV class (CV objects), one for each file that is to be uploaded. Each file is represented by one CV object.
- Run through the array to get metadata from the current file.
- Create progress messages, so that we can test if the metadata is correctly structured.

1. Create a foreach loop with the name \$CVFile in \$AllCVFiles.

```
foreach($CVFile in $AllCVFiles){
```

2. After the start curly bracket, add a new variable, CVObj, that becomes an instance of the CV class.

```
$CVObj = New-Object CV
```

3. Run the CVObj row to initialize the variable and get IntelliSense for it.
4. To add content to the foreach variable CVFile and thereby also get IntelliSense for it, enter `$CVFile = $AllCVFiles[0]` on a new row and run it.
5. Comment out or remove the code snippet `$CVFile = $AllCVFiles[0]` before you continue. Otherwise the loop will only use the first file for each iteration when you run it.

```
foreach($CVFile in $AllCVFiles){
```

```
#$CVFile = $AllCVFiles[0]
```

## 20.2.4 Get MetaData for the Decision Column

The Decision column in SharePoint will get its metadata from the subfolder names. In PowerShell a folder is called a directory.

1. To get the Decision metadata from all files, enter the CVObj variable again on a new row after the class call, and add a dot. Now you will have property suggestions from the CV class and can select Decision.
2. Set the value for the CVObj.Decision property to the folder name of the current file. (When you write a dot after `$CVFile`, you should be able to select first Directory and then Name.)

```
foreach($CVFile in $AllCVFiles){
```

```
$CVObj = New-Object CV
```

```
$CVObj.Decision = $CVFile.Directory.Name
```

3. Add the Write-Host cmdlet and `$CVObj.Decision` within a parenthesis before the loop's end curly bracket.



```
Write-Host ($CVObj.Decision)
}
```

We will add more objects to this parenthesis in later steps. Keep the Write-Host command one the row before the end curly bracket when you continue adding code to the foreach loop. The Write-Host command shows the content of the CVOB, so it should be executed last, when all the properties have been set.

4. Run the foreach loop and check that you get the three different subfolder names in the console pane: Hired, Maybe and No.

### 20.2.5 Split out the Department from the File Name

To get metadata for the three SharePoint columns First name, Last name and Department, we need to split the file names. They are written like this: LastName, FirstName - Department. When you split them, it is important to remember that PowerShell counts the first object as 0.

The first split is made at the hyphen before Department, so that we get the values for the Department column.

When we split the file names like this, we must also consider the spaces between the metadata, so that they are not included in the column values.

1. In the foreach loop, on a new row below the \$CVObj.Decision row, enter a new variable, FileNameParts with the value of an array of all file names without extensions.
2. Add the split operator and a string that contains space – space to the variable value. This will take out the department names from the file names, as they are separated from the rest of the file name with a hyphen surrounded by spaces.

```
$FileNameParts = $CVFile.BaseName -split ' - '
```

3. Initialize the new variable and check that the department part of the file name comes on a new row in the console pane. This means that the department name has been split from the rest of the file name.

```
PS C:\Users\PeterKalmström> $FileNameParts
Zubar, Annelie
Europe
```

To read the Department metadata from all files, enter the CVObj variable on a new row and add a dot. Now you will have suggestions from the CV class. Select Department and give it the value of element 1 in the FileNameParts array. (After the split, LastName, FirstName is element 0 and Department is element 1.)

```
$CVObj.Department = $FileNameParts[1]
```

4. Add the CVObj.Department object with plus operators and a space string to the parenthesis in the Write-Host command.

```
Write-Host ($CVObj.Decision + " " + $CVObj.Department)
```

5. Run the whole foreach loop again and check that you get the decisions and the department names in the console pane.

```
$FileNameParts = $CVFile.BaseName -split ' - '
$CVObj.Department = $FileNameParts[1]
Write-Host ($CVObj.Decision + " " + $CVObj.Department)
}
Hired Seattle Metro
Hired Northern
Hired Europe
Hired Eastern
Hired Seattle Metro
Hired Western
Hired Eastern
Hired Seattle Metro
```

## 20.2.6 Split the First and Last Name in the File Name

Now, we will split the file name part with the last name and the first name, to get metadata for the columns Last name and First name. This split is made at the comma between the two names, which means that LastName is 0 and FirstName is 1.

1. Enter a new variable, NameParts, and give it the value of the variable FileNameParts [0].

2. Add the `-split` operator to the variable value and split at comma space between the last and first names (to not start the first name with a space).

```
$NameParts = $FileNameParts[0] -split ', '
```

3. Initialize the `NameParts` variable and check that the last name and the first name are placed on different rows in the console pane. That means that they have been split.

```
PS C:\Users\PeterKalmström> $NameParts
Zubar
Annelie
```

4. To read the last name from all files, enter the `CVObj` variable on a new row and add a dot. Now you will have suggestions from the `CV` class. Select `LastName`.

5. Add the value of the first element in `NameParts` array.

```
$CVObj.LastName = $NameParts[0]
```

6. To read the first name from all files, enter the `CVObj` variable on a new row and select the `FirstName` property.

7. Add the value of the second element in the `NameParts` array.

```
$CVObj.FirstName = $NameParts[1]
```

8. Finally, we can convert the whole `Write-Host` object into a `CSV` string, instead of adding the two name objects within the parenthesis too, like we did above. Remove the text inside the parenthesis and add the cmdlet `ConvertTo-Csv` with value of the `CVObj` variable.

```
Write-Host (ConvertTo-Csv $CVObj)
```

9. Now the whole `foreach` loop looks like this:

10. `foreach($CVFile in $AllCVFiles){`

```
 $CVObj = New-Object CV
```

```
 $CVObj.Decision = $CVFile.Directory.Name
```

```
$FileNameParts = $CVFile.BaseName -split '-'
```

```
$CVObj.Department = $FileNameParts[1]
```

```
$NameParts = $FileNameParts[0] -split ','
```

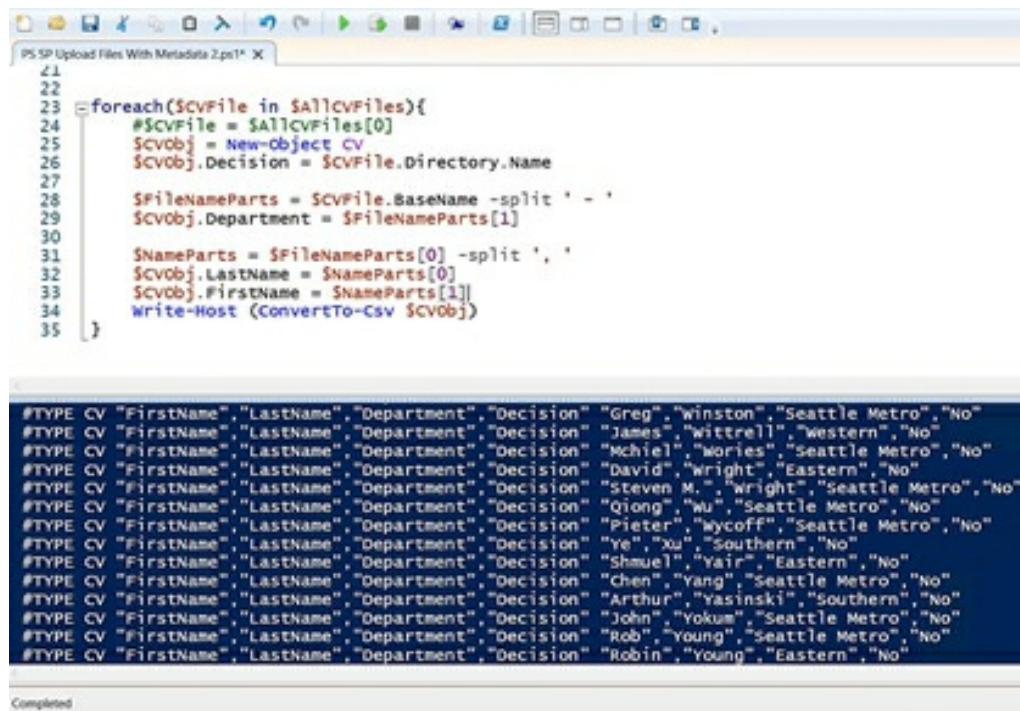
```
$CVObj.LastName = $NameParts[0]
```

```
$CVObj.FirstName = $NameParts[1]
```

```
Write-Host (ConvertTo-Csv $CVObj)
```

```
}
```

11. Run the foreach loop, and you should see the properly ordered, comma separated, metadata in the console pane.



The screenshot shows a PowerShell console window titled "PS SP Upload Files With Metadata 2.ps1 X". The script content is as follows:

```
21
22
23 foreach($CVFile in $AllCVFiles){
24 # $CVFile = $AllCVFiles[0]
25 $CVObj = New-Object CV
26 $CVObj.Decision = $CVFile.Directory.Name
27
28 $FileNameParts = $CVFile.BaseName -split '-'
29 $CVObj.Department = $FileNameParts[1]
30
31 $NameParts = $FileNameParts[0] -split ','
32 $CVObj.LastName = $NameParts[0]
33 $CVObj.FirstName = $NameParts[1]
34 Write-Host (ConvertTo-Csv $CVObj)
35 }
```

The output of the script is a list of CSV-formatted metadata for 15 different CV files. Each line starts with "#TYPE CV" followed by a header row: "FirstName", "LastName", "Department", "Decision". The data rows contain the following values:

| FirstName | LastName | Department    | Decision |
|-----------|----------|---------------|----------|
| Greg      | Winston  | Seattle Metro | No       |
| James     | Wittrell | Western       | No       |
| Mchiel    | Wories   | Seattle Metro | No       |
| David     | Wright   | Eastern       | No       |
| Steven M. | Wright   | Seattle Metro | No       |
| Qiong     | Wu       | Seattle Metro | No       |
| Pieter    | Wycoff   | Seattle Metro | No       |
| Ye        | Xu       | Southern      | No       |
| Shmuel    | Yair     | Eastern       | No       |
| Chen      | Yang     | Seattle Metro | No       |
| Arthur    | Yasinski | Southern      | No       |
| John      | Yokum    | Seattle Metro | No       |
| Rob       | Young    | Seattle Metro | No       |
| Robin     | Young    | Eastern       | No       |

The console window ends with the word "Completed".

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/160-Upload-Files-With-Metadata-2-PowerShell-SharePoint.htm>

## **20.3 SUMMARY**

In this chapter I introduced the split operator, and we have used it to split the names of the files we want to upload in three parts.

I also introduced the class, a PowerShell building block that lets us create objects with certain properties. In this chapter, we used a class to create objects with properties that correspond to the four kinds of metadata that we want to make use of when we upload files to SharePoint.

Now the metadata is structured in a way that makes it possible for us to add it to different columns in a SharePoint document library. We will do that in the next chapter.

## ***21 UPLOAD FILES WITH METADATA – UPLOAD***

In this chapter, we will expand the foreach loop that we created in the previous chapter, where the metadata from a CVs folder in Windows Explorer was structured.

When we run the finished foreach loop, the files will be uploaded and their metadata set in the four columns we created in the CVs SharePoint document library in chapter 19.

### **21.1 THEORY**

We will be using the Add-PnPFile cmdlet to upload the files, just as we did in chapter 16. However, as we now want to include metadata in the upload, we need to add a new parameter to the cmdlet: Value.

It is possible to add all the values in a long string after the Value parameter, but for clarity and better readability, we will instead create a hashtable with key-value pairs. I introduced both key-value pairs and the Value parameter in chapter 14.

In this exercise, we will use the += operator in the same way as we did in chapter 15: to add another key-value pair to the first one. We will add such key-value pairs for all the four columns that we want to fill with metadata.

The keys will be the internal names for the SharePoint columns we created in chapter 19, and the values will be the four metadata object properties we created in the previous chapter.

### **21.2 STEPS**

Continue adding code to the foreach loop we created in the previous chapter, to get it to upload the files to SharePoint also, and not just structure the metadata.

We will first create the hashtable of key-value pairs, and then we will add an Add-PnPFile command that uploads the files and reads the metadata distribution from that hashtable.

Before you start adding to the loop, comment out or remove the Write-Host cmdlet. We don't need that row anymore, because the file upload will be visible in the console pane anyway. With the Write-Host command we will have double messages, and even if it does not affect the upload to SharePoint it will look messy.

1. Enter the new variable name CVValues on the first empty row within the foreach loop's curly brackets (after the commented out Write-Host command).

2. Add the value of an empty hashtable.

```
$CVValues = @{ }
```

3. To have the file names in the SharePoint Title column, enter the CVValues variable with the added value of the key-value pair Title and the BaseName property of the current file (without file extension).

```
$CVValues += @{'Title'=$CVFile.BaseName }
```

4. To have the decision values in the SharePoint Decision column, enter the CVValues variable with the added value of the key-value pair CVDecision and the Decision property of the current object.

```
$CVValues += @{'CVDecision'=$CVObj.Decision}
```

5. Continue in the same way to add the values for the First name, Last name and Department columns.

```
$CVValues += @{'CVFirstName'=$CVObj.FirstName }
```

```
$CVValues += @{'CVLastName'=$CVObj.LastName}
```

```
$CVValues += @{'CVDepartment'=$CVObj.Department}
```

6. Enter the Add-PnPFile cmdlet with the parameters Path and File.

The Path value should be the current file name. Use the FullName property to include the extension. (The FullName property contains the full path, including the folder path.)

The Folder value should be the name of the SharePoint library, here CVs.

7. Add the Value parameter to the Add-PnPFile cmdlet and give it the value

of the CVValues hashtable.

```
Add-PnPFile -Path $CVFile.FullName -Folder "CVs" -Values $CVValues
```

## 8. Run the whole foreach loop.

```
foreach($CVFile in $AllCVFiles){
 # $CVFile = $AllCVFiles[0]

 $CVObj = New-Object CV

 $CVObj.Decision = $CVFile.Directory.Name

 $FileNameParts = $CVFile.BaseName -split ' - '

 $CVObj.Department = $FileNameParts[1]

 $NameParts = $FileNameParts[0] -split ', '

 $CVObj.LastName = $NameParts[0]

 $CVObj.FirstName = $NameParts[1]

 #Write-Host (ConvertTo-Csv $CVObj)

 $CVValues = @{}
 $CVValues += @{"Title"=$CVFile.BaseName }
 $CVValues += @{"CVDecision"=$CVObj.Decision}
 $CVValues += @{"CVFirstName"=$CVObj.FirstName }
 $CVValues += @{"CVLastName"=$CVObj.LastName}
 $CVValues += @{"CVDepartment"=$CVObj.Department}
 Add-PnPFile -Path $CVFile.FullName -Folder "CVs" -Values $CVValues
}
```

## 9. Check that the files have been uploaded to the SharePoint library and that



the metadata is correctly set in the library columns.



| Name                                | Decision | First Name | Last Name   | Department    |
|-------------------------------------|----------|------------|-------------|---------------|
| Aaberg, Jesper - Seattle Metro.docx | Hired    | Jesper     | Aaberg      | Seattle Metro |
| Aalling, Lene - Northern.docx       | Hired    | Lene       | Aalling     | Northern      |
| Abbas, Syed - Europe.docx           | Hired    | Syed       | Abbas       | Europe        |
| Abercrombie, Kim - Eastern.docx     | Hired    | Kim        | Abercrombie | Eastern       |
| Abola, Lina - Seattle Metro.docx    | Hired    | Lina       | Abola       | Seattle Metro |

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/170-Upload-Files-With-Metadata-3-PowerShell-SharePoint.htm>

## 21.3 SUMMARY

This was the third chapter about uploading files with metadata, and we have reached our goal. The files are uploaded to SharePoint, and the metadata for each file is added to the correct column.

In the process, you have learned new cmdlets and parameters, and you have seen how a PowerShell class can be used to add properties to objects. We have split file names in three parts and made use of these parts as values for the metadata objects, and we have once again worked with key value pairs and the addition operator +=.

## ***22 CREATE A SHAREPOINT INTRANET - SUBSITES***

In the last eight chapters of the book, I have chosen to create an intranet for a small business. It will have three subsites, one for each department.

The subsites and the root site should have a common navigation, but each subsite will have its own theme and apps where team members can share documents, photos, appointments and tasks.

Each subsite will also have three site pages, and we will add a YouTube video on one of the homepages.

I chose this specific business scenario as my final example, because it is the topic of my most popular YouTube video. In the video I create the whole intranet with 3 department sites in about 13 minutes. To do it with PowerShell takes longer, but I hope these chapters will illustrate some useful real world demands.

Also, if you need to create multiple site collections like this, a script will of course save time for you!

### **22.1 PREREQUISITES**

We both created and removed site collections for five departments in chapter 13. For this exercise we will use one such collection, a HQ Communication site.

You can create a single site collection with a simple `New-PnPSite` command. Just select the site type and hardcode the Title and Url parameters.

```
New-PnPSite -Type CommunicationSite -Title HQ -Url "https://.../sites/HQ"
```

This HQ site will be our starting point, and the URL variable in the script below points to that site.

### **22.2 CREATE SUBSITES**

We will create three subsites to the HQ site, for the Sales, Production and

Support departments. These sites will have the same name as the department, and they will also be Communication sites.

## 22.2.1 Theory

In chapter 13, we saw that the cmdlets `New-PnPSite` and `New-PnPTenantSite` create a new site collection. When we want to create a subsite, we must use another cmdlet, because the SharePoint UI and the PowerShell API use the term "site" differently.

In PowerShell, a site means a site collection. Sites inside a site collection are called webs. Therefore, we cannot use the same cmdlet to create the subsites as when we create the root site of a site collection.

(In the SharePoint UI all kinds of sites are called sites, and the word web only shows up here and there sporadically.)

Generally, we must use cmdlets with "Site" when we are working with site collections and cmdlets with "Web" in connection with subsites.

When we want to add subsites, we should use the cmdlet `New-PnPWeb`. This cmdlet creates a subsite with some default apps depending on site template, within the site collection that PowerShell is connected to.

One of the mandatory parameters for subsite creation with `New-PnPWeb` is `template`. On this page, you can find the ID for many different site templates: <https://www.technologytobusiness.com/microsoft-SharePoint/sharepoint-online-site-template-id>

We will use the `Template` value `SITEPAGEPUBLISHING#0` for a Communication site.

(This might seem confusing because in chapter 14 we used `CommunicationSite` for a Communication site. That is because the cmdlet `New-PnPSite` takes the parameter `Type`, not `Template`. The `Type` value `CommunicationSite` actually has the same ID as the `Template` value used by the `New-PnPWeb` cmdlet.)

By default, everything created by the `New-PnPWeb` cmdlet inherits the security settings from the parent site. Should you want to break the inheritance, use the parameter `BreakInheritance`.

## 22.2.2 Steps

We will create an array for the three departments, and then we will build a foreach loop that runs through the array and creates subsites with the department names.

1. Create an array called Departments that contains the three department names.

```
$Departments = "Sales","Production","Support"
```

2. Initialize the variable.
3. Create a foreach loop that runs through the Departments array.
4. After the first curly bracket, add the cmdlet New-PnPWeb to create subsites.
5. Add the Title and URL parameters and give them both the value of the current department name.
6. Add the InheritNavigation parameter.
7. Add the Template parameter and give it the value for the Communication site.

```
foreach($Dep in $Departments){
 New-PnPWeb -Title $Dep -Url $Dep -InheritNavigation -Template
 SITEPAGEPUBLISHING#0
```

8. Run the foreach loop and check that the three subsites have been created under the HQ site in SharePoint.

## 22.3 REMOVE SUBSITES

To be able to test and recreate subsites, we need a quick way to remove them.

### 22.3.1 Theory

One of the advantages with PowerShell is that it is easy to remove earlier trials and test a script again.

In this exercise, we are connecting to the root site of the site collection with the URL variable.

In the following chapters, we will also connect to the subsites, and we will do that with the variable CurrWebURL.

To remove the subsites, you must make sure that you are connected to the root site and not to the subsites, because you cannot remove subsites if you are not connected to the root site. This is something you must always consider when you work with multiple sites.

To be sure the connection gets right when you run the whole script, you can add connection commands first in each block of code that requires a change of URL.

### 22.3.2 Steps

For the removal, we will use the same Departments array as in the creation foreach loop above.

1. Create a foreach loop that runs through the Departments array.
2. Add the cmdlet Remove-PnPWeb after the start curly bracket.
3. Add the URL parameter and value.
4. (Add a Force parameter if you don't want to be asked about each removal.)

```
5. foreach($Dep in $Departments){
 Remove-PnPWeb -Url $Dep -Force
}
```

6. Run the foreach loop and check that the subsites are removed from SharePoint.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/990-Small-Bussiness-Intranet-1-Create-Subsites-PowerShell-SharePoint.htm>

## 22.4 SUMMARY

In this chapter, we have first created a new site collection with a New-PnPSite command, and then we have added three subsites using the cmdlet New-PnPWeb.

The New-PnPWeb command was used by a foreach loop that runs through an array of three department names. The array was also used in another foreach loop that removes the subsites again, and we will use the same array in the following chapters, when we expand our small intranet.

In the next exercise, we will add a common navigation to all the four sites.

## ***23 CREATE A SHAREPOINT INTRANET - NAVIGATION***

In this chapter, we will enhance the SharePoint intranet we created in the previous chapter with a common navigation. The navigation will also have a link to an external site. For this link we will use the External parameter, which will make the link open in a new tab.

The removal foreach loop that we created in the previous chapter does not remove the external link on the HQ site. For that we must use another cmdlet and another loop.

### **23.1 THEORY**

The subsites created with the New-PnPWeb cmdlet does not inherit the navigation from the parent side, so we added the parameter InheritNavigation.

However, the InheritNavigation parameter is not enough to create the common navigation we want for all the sites. We must also specify the target URL of each site.

Note that the Communication sites only have the local navigation – the Quick launch – and it is placed on top, where other site types have their global navigation.

The intranet navigation will be added in three steps, with some check runs in between:

- The code for the subsite links on the root site is added to the foreach loop we created in the previous chapter.
- When we add links to the subsites, we need to loop through them. Therefore, the process must be performed by a second foreach loop that connects to each subsite.

The subsite URL is the URL of the root site with the addition of a slash and the department name. This URL value is written within a parenthesis, to be

calculated first, and with addition operators.

`($URL + "/" + $Dep)`

The slash must be written within quotation marks to explain that it should be interpreted as a string and not as an operator.

- The external link on the root site can be added with just an `Add-PnPNavigationNode` command. However, as we earlier connected to the subsites, we must first connect back to the root site again.

This connecting back might cause a time-out error when the whole script is run. Therefore, we will add a `Start-Sleep` command that suspends the activity for five seconds.

`Start-Sleep -Seconds 5`

## 23.2 STEPS

We will add code to the `foreach` loop for subsite creation that we built in the previous chapter, so before you start adding the navigation, make sure that you have used the removal script and cleared away the subsites. If you don't do that, you will get an error when you run the loop, because the subsites already exist.

Another option, if you have not removed the subsites already, is to just comment out the subsite creation command before you run the `foreach` loop to check the navigation.

### 23.2.1 Subsite links on the HQ site

We will start by adding subsite links on the root site (HQ).

1. Enter a new cmdlet, `Add-PnPNavigationNode`, before the end curly bracket in the `foreach` loop.
2. Add the parameter `Location` with the value `QuickLaunch`.
3. Add the parameter `Title` with the value of the current department name.
4. Add the parameter `Url` with the value of the `URL` variable / current



department name.

```
Add-PnPNavigationNode -Location QuickLaunch -Title $Dep -Url ($URL + "/" + $Dep)
}
```

5. Now the whole foreach loop looks like this:

```
foreach($Dep in $Departments){
 New-PnPWeb -Title $Dep -Url $Dep -InheritNavigation -Template
 SITEPAGEPUBLISHING#0

 Add-PnPNavigationNode -Location QuickLaunch -Title $Dep -Url ($URL + "/" + $Dep)
}
```

6. Run the loop and check in SharePoint that the subsites have been created again and that links to them have been added to the HQ site Quick launch.

### 23.2.2 Links on the subsites

Now we will work with the subsites, so the new foreach loop that we will create here must start the execution with connecting to the current subsite.

Each of the subsites will have links to the other subsites, to the HQ site and to kalmstrom.com.

1. Create another foreach loop for the Departments array.
2. After the start curly bracket, enter a new variable, with the name CurrWebURL and the value of the URL to the current subsite, which we have already defined in the earlier foreach loop.

```
foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
```

3. On a new row, enter the Connect-PnPOnline cmdlet and use the value of the CurrWebURL variable for the Url parameter (which is not written out here).

```
Connect-PnPOnline $CurrWebURL
```

4. On the next row and still inside the foreach loop's curly brackets, enter

another foreach loop, that runs through the subsites and add links to the other subsites. Name the current subsite variable DepSub, to distinguish this loop from the superior one.

```
foreach($DepSub in $Departments){
```

5. The new foreach loop should not add a link to the current subsite in the Quick launch of that same subsite (for example, the Sales subsite should not have a link to Sales).

To avoid that, enter an If statement so that the interior foreach loop. only runs if the current department name is not equal to the current subsite name).

```
if($Dep -ne $DepSub){
```

6. Enter the cmdlet Add-PnPNavigationNode after the If statement start curly bracket.
7. Add the Location parameter with the value QuickLaunch and the Title with the value of the current subsite.
8. For the target link, add the parameter Url with the value of the URL variable / current subsite.

```
Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" + $DepSub)
```

9. End the If statement and the interior foreach loop with two end curly brackets.
10. After the end curly bracket of the interior foreach loop, enter another Add-PnPNavigationNode cmdlet with the Location value Quick launch, the Title value HQ and the Url value of the URL variable. That will give the subsites a link to the HQ site.

```
Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL
```

Add another Add-PnPNavigationNode cmdlet with the Location Quick launch, the Title kalmstrom.com and the Url value <https://www.kalmstrom.com>. Also add the External parameter to this

cmdlet.

Now the foreach loop looks like this:

```
foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 Connect-PnPOnline $CurrWebURL

 foreach($DepSub in $Departments){
 if($Dep -ne $DepSub){
 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" +
$DepSub)
 }
 }
 Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL

 Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External
}
```

11. Run the whole foreach loop and check in SharePoint that the subsites have links to the other subsites and to the HQ site and that all subsites have an external link to kalmstrom.com.

### 23.2.3 Link to an external site on the HQ sited

We added links to an external site in the second foreach loop, but we also need to add that link on the HQ site.

As we connected to the current subsite in the steps above, we now must connect to the HQ site again. We will do that with a five seconds delay to avoid a timeout issue. Then we can add the link with a Connect-PnPOnline command.

1. Enter the Start-Sleep cmdlet on a new row below the second foreach loop.

2. Add the parameter Seconds and the value 5.
3. Copy the row Connect-PnPOnline \$URL on top of the script and paste it to the row below the Start-Sleep command.
4. Copy the external link row in the second foreach loop and paste it below the Connect-PnPOnline command.

```
Start-Sleep -Seconds 5
```

```
Connect-PnPOnline $URL
```

```
Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
```

```
"https://kalmstrom.com" -External
```

5. Run the three new lines of code, to connect to the HQ site and add the kalmstrom.com link to that site also.
6. Check in SharePoint that the HQ site has the external link.

Now the whole block of navigation code looks like this:

```
$Departments = "Sales","Production","Support"
```

```
foreach($Dep in $Departments){
```

```
 New-PnPWeb -Title $Dep -Url $Dep -InheritNavigation -Template SITEPAGEPUBLISHING#0
```

```
 Add-PnPNavigationNode -Location QuickLaunch -Title $Dep -Url ($URL + "/" + $Dep)
```

```
}
```

```
foreach($Dep in $Departments){
```

```
 $CurrWebURL =($URL + "/" + $Dep)
```

```
 Connect-PnPOnline $CurrWebURL
```

```
foreach($DepSub in $Departments){
```

```
 if($Dep -ne $DepSub){
```

```
 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" + $DepSub)
```

```

 }
}

Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL

Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External
}

Start-Sleep -Seconds 5

Connect-PnPOnline $URL

Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External

```

7. Run the removal foreach loop that we created in chapter 22 to remove all the subsites again. This will also remove the links to them in the HQ site, but it will not remove the external link on the HQ site, see the section below.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/991-Small-Bussiness-Intranet-2-Navigation-PowerShell-SharePoint.htm>

## 23.3 REMOVE EXTERNAL LINK

In the previous chapter, we created a foreach loop that removes the subsites and the links from the root site to these subsites. However, that loop does not remove the external link on the HQ site, so we need to create a new foreach loop for that removal.

### 23.3.1 Theory

To remove a link, we need to use the cmdlet `Remove-PnPNavigationNode`. This cmdlet requires the value of the `Identity` parameter, but that is something

we don't know.

To solve the problem, we will create a foreach loop that runs through an array of all navigation nodes and removes the one that has the same title as the external link. As we have IntelliSense for the properties of the node objects, we can select the Id property without knowing the actual ID!

### 23.3.2 Steps

1. On a new row below the earlier removal foreach loop, create the variable `AllNodes` for all the navigation nodes.
2. Give the variable the value of an array created by the `Get-NavigationNode` cmdlet with the parameter `Location` and the value `QuickLaunch`.

```
$AllNodes = Get-PnPNavigationNode -Location QuickLaunch
```

3. Initialize the variable.
4. On the row below the variable, enter a foreach loop that runs through the `AllNodes` array.
5. To get IntelliSense for the `Node` variable, add a new row after the loop's start curly bracket and give the `Node` variable the value of the first item in the `AllNodes [0]` array.

```
foreach($Node in $AllNodes){
```

```
 $Node = $AllNodes [0]
```

6. Run the last row and then comment out or remove it.
7. Add an If statement after the loop's start curly bracket: if the current navigation node's `Title` property equals the `Title` value of the external link.

```
if($Node.Title -eq "kalmstrom.com"){
```

8. After the If statement's start curly bracket, enter the cmdlet `Remove-PnPNavigationNode`.
9. Add the `Identity` parameter and set its value to the `Node` variable with the property `Id`. Here I have also added the `Force` parameter.

```
$AllNodes = Get-PnPNavigationNode -Location QuickLaunch
foreach($Node in $AllNodes){
 if($Node.Title -eq "kalmstrom.com"){
 Remove-PnPNavigationNode -Identity $Node.Id -Force
 }
}
```

10. Run the foreach loop and check that the kalmstrom.com link has been removed from the HQ site.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/992-Small-Bussiness-Intranet-3-Apps-PowerShell-SharePoint.htm> – the first part

## 23.4 SUMMARY

In this exercise we have used an encapsulated foreach loop for the first time. This is useful if you want to set a condition for part of the loop, as we did here.

We have also switched between connections for the first time, and as I said already in the previous chapter, this can create problems if you are using the wrong connection.

PowerShell ISE can be very helpful with the IntelliSense suggestions, and in this chapter, we have been able to set the value of the Identity parameter without knowing the actual ID.

In the next chapter we will add apps to the subsites. You already know how to create apps in different ways. Here we will add a call to the function Create-MyApp in the foreach loop that is connected to the subsites.

## 24 CREATE A SHAREPOINT INTRANET – APPS

Now it is time to create apps for the three department sites in our intranet. When we create a Communication site, a document library and a calendar are created automatically. Now we will add two more apps: a Tasks list and a Picture library.

I used app creation in the early chapters of this book to introduce important PowerShell concepts, so this chapter will be a repetition.

### 24.1 THEORY

There is not much new theory in this chapter, but you might want to look back on chapter 12. As we are adding two types of apps to the subsite here, we will call the general Create-MyApp function that we built in that chapter.

```
function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template $AppType -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
 Green
}
```

The apps will be created in the subsites, so the two function calls should be added to the second foreach loop, the one that we built to create the subsite navigation in chapter 23.

### 24.2 STEPS

Make sure that you have used the Remove function we created in the



previous chapter, so that you can start with a clean HQ site. Everything will be created again, as we are going to run the whole code.

We will create the apps by calling the Create-MyApp function and giving the applicable parameters and values for the two app types. The two calls must be placed in the second foreach loop, and the app creation should be the first thing that is executed after the connection to the subsite.

1. Enter a call to the Create-MyApp function on the row below the Connect-PnPOnline command.

2. Add the parameters ListName and AppType with the values "Photos" and PictureLibrary.

```
Create-MyApp -ListName "Photos" -AppType PictureLibrary
```

3. Enter another Create-MyApp call.

4. Add the parameters ListName and AppType with the values "Tasks" and Tasks.

5. Now the second foreach loop looks like this:

```
foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 Connect-PnPOnline $CurrWebURL

 Create-MyApp -ListName "Photos" -AppType PictureLibrary

 Create-MyApp -ListName "Tasks" -AppType Tasks

 foreach($DepSub in $Departments){
 if($Dep -ne $DepSub){
 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" +
$DepSub)
 }
 }
}
```

```
Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL
```

```
Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External
}
```

6. Run the whole intranet creation code, from the Departments array and down to the external link on the HQ site.
7. Check that the subsites and the navigation has been added again and that each subsite has Photos and Tasks apps.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/992-Small-Bussiness-Intranet-3-Apps-PowerShell-SharePoint.htm> – the second part

## 24.3 SUMMARY

This chapter was just a repetition of app creation, so I hope you managed it quickly. We just added two function calls to the foreach loop, and when the loop now runs through the Departments array the two apps will be added to each department's subsite.

In the next chapter we will not add anything new to the intranet. Instead, we will clean up the code by moving parts of it into functions.

## ***25 CREATE A SHAREPOINT INTRANET – FUNCTIONS***

In this chapter, we will come back to function building, because we will add some of the code we have created for the intranet into functions. This will make the code easier to overview, and we will only have to call a function when we want to run a piece of script, instead of repeating the whole execution code.

### **25.1 THEORY**

In the previous chapter, we ended up with a block of code for subsites with apps and common navigation that is rather difficult to read. We can make the code look better by moving some of it into functions and just have function calls in the foreach loop.

Experienced coders of course write the functions directly, but here we will cut out blocks of code that we have created earlier and paste them in new functions. I hope that will make it easier to understand how functions work. In chapter 28 we will create a new function in the proper way, without cut and paste.

Functions have several advantages:

- The code becomes easier to overview, as functions can be collapsed.
- It is easier to call a function than to run a long execution code.
- It is quicker to comment out a function call than commenting out all the code that is contained in the function.
- As functions are executed from calls, a function can stay untouched in the script even if you don't want to use it right now. You just need to comment out or remove the call.
- For critical functions, like removals, you can choose to only call the function at the command prompt and never include a call in the script. Therefore, will add our two pieces of removal code into one function.

Even if functions are very useful, you must be careful when you choose the names for your functions. The name must give a good description of which process the function executes, so that it is easy to understand what the function does without expanding the function and checking the code.

## 25.2 ADD REMOVAL CODE TO FUNCTION

Currently we have two foreach loops for removal:

- a Remove-PnPWeb loop that takes away all the subsites and the links to these subsites on the HQ site.
- a Remove-PnPNavigationNode loop that takes away the link to the external site from the HQ site.

We will now create a function and add these two loops to it. We will call the function Remove-All.

When we run the whole script, it will connect back from the subsites to the root site in the very last step. However, if we want to run only the removal code, that connection will not be included, and if we are connected to the subsites, the removal will not work. Therefore, we will also add a connection to the root site in the Remove-All function.

### 25.2.1 Steps

It is suitable to place the removal function last in the script.

1. Start creating a function with the name Remove-All at the bottom of the script pane.
2. Add an empty parenthesis after the function name to show that there are no input parameters. This is not strictly mandatory but is often done.
3. Cut the code that connects to the root site and add paste it after the loop's start curly bracket.
4. Cut the foreach loop with the Remove-PnPWeb command and paste it after the connection command.

5. Cut the line of code with the AllNodes variable and paste it after the end curly bracket of the foreach loop.
6. Cut the foreach loop with the Remove-PnPNavigationNode command and paste it after the AllNodes variable, before the function's end curly bracket.
7. Make sure you have ended all the curly brackets.

```
function Remove-All(){
 Connect-PnPOnline $URL
 foreach($Dep in $Departments){
 Remove-PnPWeb -Url $Dep -Force
 }
 $AllNodes = Get-PnPNavigationNode -Location
QuickLaunch
 foreach($Node in $AllNodes){
 if($Node.Title -eq "kalmstrom.com"){
 Remove-PnPNavigationNode -Identity $Node.Id -Force
 }
 }
}
```

8. Initialize the function.
9. Call the function at the command prompt to remove all the subsites and links.
10. Minimize the function.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/992-Small-Bussiness-Intranet-3-Apps-PowerShell-SharePoint.htm> - the first part

## 25.3 ADD APP AND NAVIGATION CODE TO FUNCTION

The foreach loop we created for subsite navigation and apps in the chapters

23 and 24 is rather long and complicated:

```
foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 Connect-PnPOnline $CurrWebURL

 Create-MyApp -ListName "Photos" -AppType PictureLibrary

 Create-MyApp -ListName "Tasks" -AppType Tasks

 foreach($DepSub in $Departments){
 if($Dep -ne $DepSub){
 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" +
$DepSub)
 }
 }

 Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL

 Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External
}
```

When this exercise is finished, the foreach loop will instead look like this:

```
foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 Connect-PnPOnline $CurrWebURL
```

```
Add-Apps
```

```
Add-SubsiteNavigation
```

```
}
```

As you see, two function calls have replaced whole part after the connection to the subsites. All the code that add apps and navigation to the subsites has been moved into two functions.

When this is done, we will run the script and make sure that everything is created again.

### 25.3.1 Steps

Create the functions above the foreach loop that will call these functions.

1. Start creating a function with the name Add-Apps. It should not have any input parameters.
2. Cut the two Create-My-App calls for Photos and Tasks from the foreach loop.
3. Paste the Create-My-App calls between the Add-Apps function's curly brackets.

```
function Add-Apps(){
 Create-MyApp -ListName "Photos" -AppType PictureLibrary

 Create-MyApp -ListName "Tasks" -AppType Tasks
}
```

4. In the foreach loop, on the place where you cut the Create-My-App commands, enter a call to the Add-Apps function.
5. Initialize the function and collapse it.
6. Enter another function below the Add-Apps function. Give it the name Add-SubsiteNavigation. It should not have any input parameters.
7. Cut the encapsulated foreach loop and the two Add-PnPNavigationNode commands from the foreach loop.
8. Paste everything between the Add-SubsiteNavigation function's curly brackets.

```
function Add-SubsiteNavigation(){
 foreach($DepSub in $Departments){
 if($Dep -ne $DepSub){
```

```

 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -Url ($URL + "/" +
$DepSub)
 }
}

Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL

Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -Url
"https://kalmstrom.com" -External
}

```

9. Add a call to the Add-SubsiteNavigation function in the foreach loop where you cut the navigation code.
10. Initialize the new function and collapse it.
11. Run the whole script and check that everything gets created again in SharePoint.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/993-Small-Bussiness-Intranet-4-Pages-PowerShell-SharePoint.htm> – the first part

## 25.4 SUMMARY

Now you should have a nicer script with three new functions, and when you have collapsed the function the code is both shorter and easier to overview. I hope I have managed to transfer my enthusiasm for functions in this chapter!

In the next chapter, we will extend the intranet again. The subsites will have three new pages each: Progress, Problems and Plans. We will create modern pages in the exercise, but I will also explain how to create classic wiki pages.



## ***26 CREATE A SHAREPOINT INTRANET – PAGES***

In chapter 24, we added apps to the three department subsites, and now it is time to give them some pages. A Site pages library was created automatically when we created the Communication subsites, so we only need to create the new pages.

Each department will have three pages: Progress, Problems and Plans. These will be modern SharePoint pages.

### **26.1 THEORY**

There are several cmdlets for pages, and you can find information about them all in the command panes. However, the most common scenario is that you want to create a new modern or classic page, and we are going to create a modern page in this exercise.

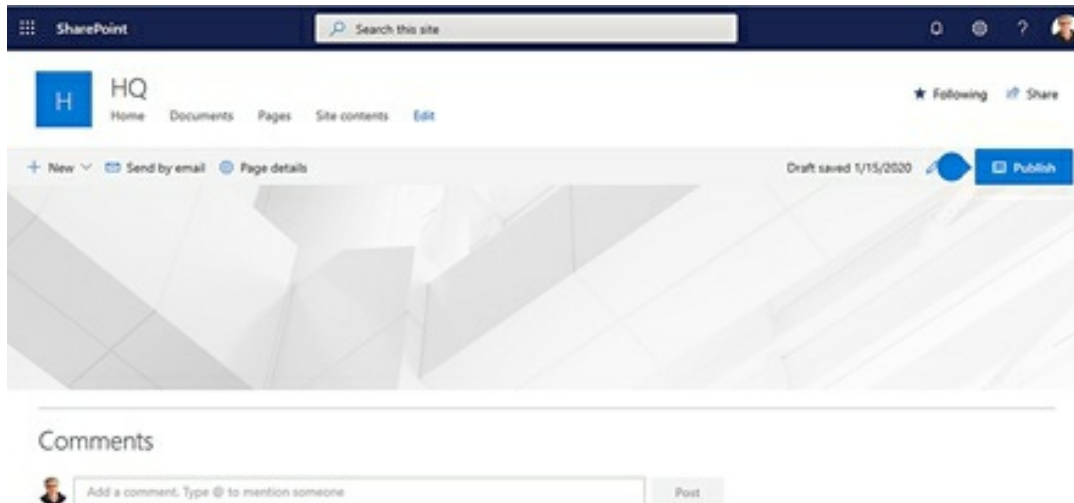
Should you prefer classic pages, you can just replace the command for all three or some of the pages in this exercise with the classic option I describe below.

#### **26.1.1 Modern Page**

The cmdlet for a modern page is `Add-PnPClientSidePage`, and it only requires one parameter: `Name`.

```
Add-PnPClientSidePage -Name "Progress"
```

This cmdlet will create a modern page within the site you are connected to.



## 26.1.2 Classic Page

To create a classic page instead, use the cmdlet `Add-PnPWikiPage`. This will give you the most used classic page, with the best customization possibilities.

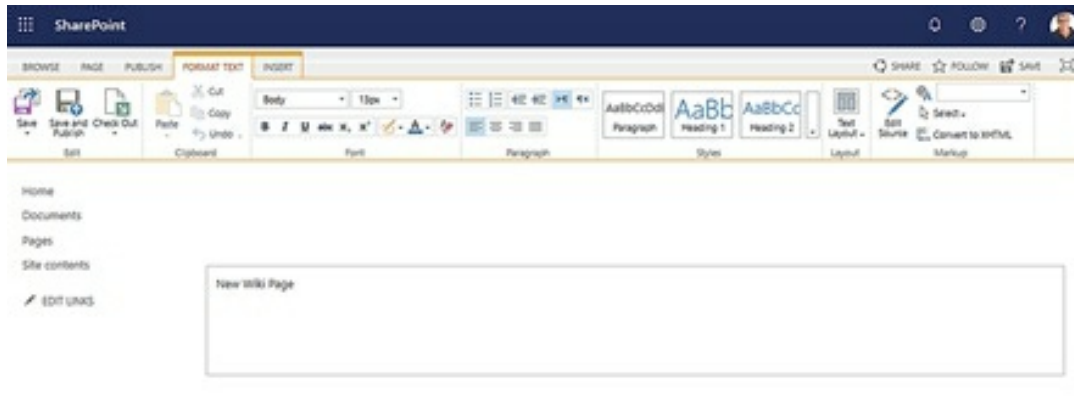
The `Add-PnPWikiPage` cmdlet always requires a server relative URL. As the page name is given in the URL here, there is no `Name` parameter.

Besides that, there are two alternatives for this cmdlet. You can either use a `Content` parameter or a `Layout` parameter.

For `Content`, you can add the value `New Wiki Page`.

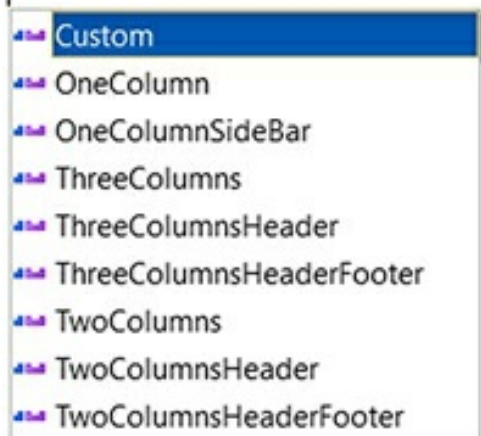
```
Add-PnPWikiPage -ServerRelativePageUrl /sites/Sales/SitePages/Plans.aspx -Content "New Wiki Page"
```

This alternative will give you a blank wiki page, that you can customize as you wish.



The Layout alternative lets you choose a layout for the page.

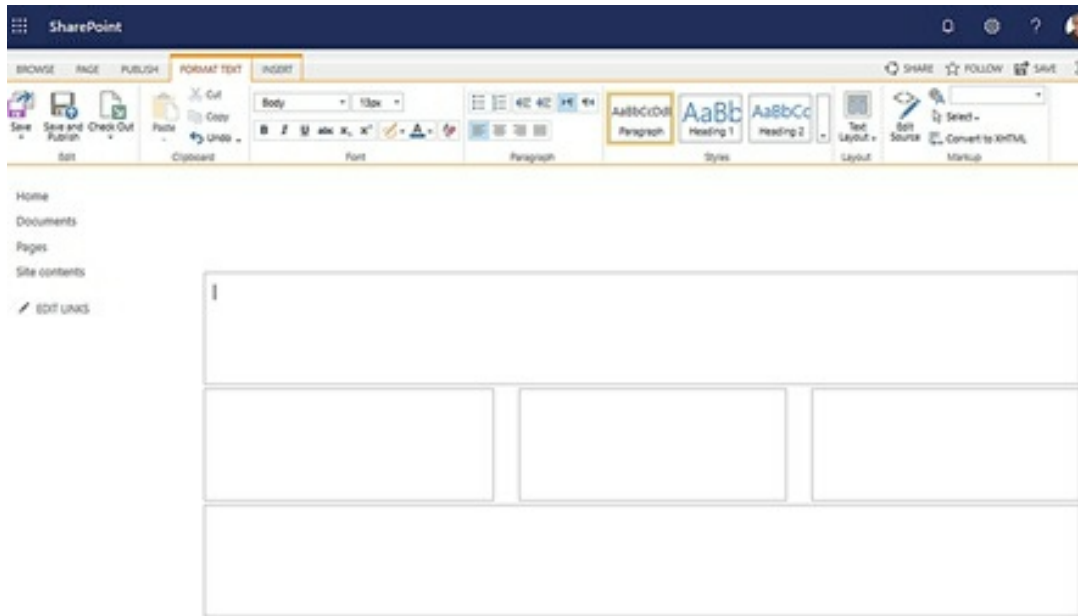
-Layout



Here I have selected the most complicated one!

[Add-PnWikiPage -ServerRelativePageUrl /sites/Sales1/SitePages/Plans.aspx -Layout](#)

[ThreeColumnsHeaderFooter](#)



## 26.2 STEPS

To create the pages in the subsites, we will continue adding code to the foreach loop that runs through the subsites and that we simplified in the previous chapter.

We will first create the three pages and make sure that the code works. After that, we will create a new Add-Pages function and add a call to it from the foreach loop.

Finally, we will remove all the subsites and links by calling the Remove-All function, so that we can create them again in the next chapter.

In the foreach loop, start on a new row below the call to the Add-SubsiteNavigation function.

1. Enter the cmdlet Add-PnPClientSidePage.
2. Add the Name parameter with the value Progress.
3. Comment out the two function calls above the Add-PnPClientSidePage command, so that you can try the command without getting errors because the apps and the navigation already exist.
4. Run the foreach loop and check in SharePoint that the Progress page has

been created.

5. Copy the Add-PnPClientSidePage command and paste it twice on the rows below.

6. Change the Name values into Problems and Plans.

```
Add-PnPClientSidePage -Name "Progress"
```

```
Add-PnPClientSidePage -Name "Problems"
```

```
Add-PnPClientSidePage -Name "Plans"
```

7. Comment out the Progress command too and then run the loop again. Make sure that the other two pages have been created in SharePoint.

8. Remove the hashtags that commented out the calls and the first page creation, so that they are included in the loop again.

9. Create a new function above the foreach loop and call it Add-Pages. It should not have any input parameters.

10. Cut the three Add-PnPClientSidePage commands and paste them within the curly brackets of the new function.

```
function Add-Pages(){
 Add-PnPClientSidePage -Name "Progress"

 Add-PnPClientSidePage -Name "Problems"

 Add-PnPClientSidePage -Name "Plans"
}
```

11. Initialize the function and then collapse it.

12. Add a call to the Add-Pages function below the other calls in the foreach loop.

13. Now the foreach loop that runs through the subsites looks like this:

```
foreach($Dep in $Departments){
```

```
$CurrWebURL =($URL + "/" + $Dep)
```

```
#Connect to the department site
```

```
Connect-PnPOnline $CurrWebURL
```

```
Add-Apps
```

```
Add-SubsiteNavigation
```

```
Add-Pages
```

```
}
```

14. Run a call to the Remove-All function at the command prompt and check that the subsites and all the links are gone from SharePoint.

```
PS C:\Users\PeterKalmström>> Connect-PnPOnline $URL
PS C:\Users\PeterKalmström>> Remove-All
```

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/993-Small-Bussiness-Intranet-4-Pages-PowerShell-SharePoint.htm> – the second part

## 26.3 SUMMARY

In this chapter we built a function that adds pages in the three department subsites when we call it. The call was of course added to the other calls in the foreach loop that works with the subsites.

We created modern pages with the cmdlet Add-PnPClientSidePage here, but I also explained how to create a classic wiki page.

In the next chapter, we will add progress messages to the script. That will make it easier to see in the console pane what is happening when the script is run.

## ***27 CREATE A SHAREPOINT INTRANET – PROGRESS MESSAGES***

If you have followed my instructions in the previous chapters, you have created and removed subsites several times. Maybe you have asked yourself what is actually happening when the script is running?

We could have added progress messages for the intranet in each chapter, just as we have done in earlier exercises, but this time I wanted you to really see the difference these messages make.

Therefore, in this chapter we will add progress messages to the intranet creation script that we have created so far.

### **27.1 THEORY**

We want to add progress messages to the parts of the script for intranet creation that takes some time to run through, and we will do that with the same cmdlet as we have used earlier: Write-Host.

Remember this is all progress messages:

- Add addition operators between the different parts of the progress message.
- Indicate which parts of the message that should be considered strings.
- Put the message inside a parenthesis.

I have used yellow color and three dots for the "creating" messages and green color for the "done" messages. Below, you can see how we will add progress messages to the foreach loop that creates subsites and adds navigation to the root site. I have made the colors bold.

```
foreach($Dep in $Departments){
 Write-Host ("Creating subsite " + $Dep + " ...") -ForegroundColor Yellow

 New-PnPWeb -Title $Dep -Url $Dep -InheritNavigation -Template SITEPAGEPUBLISHING#0

 #Add navigation node to the top site
```

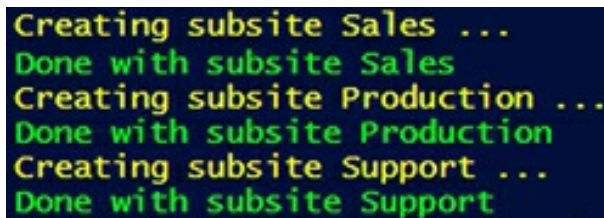
```

Add-PnPNavigationNode -Location QuickLaunch -Title $Dep -Url ($URL + "/" + $Dep)
 Write-Host ("Done with subsite " + $Dep) -ForegroundColor Green
}

```

The foreach loop above gives the first progress message in yellow when it has started to create the first subsite. The second message is green and comes when that subsite has been created.

The third message comes when PowerShell has started to create the second site, and the fourth message comes when that subsite has been created. And so on ...



```

Creating subsite Sales ...
Done with subsite Sales
Creating subsite Production ...
Done with subsite Production
Creating subsite Support ...
Done with subsite Support

```

In the second foreach loop, that adds navigation, apps and pages to the subsites, it is important to connect to each subsite. Therefore, we will give a message about that.

```

foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 #Connect to the department site

 Write-Host ("Fixing " + $CurrWebURL + " ... ") -ForegroundColor Yellow
}

```

When the connection is established, the loop creates apps with a function call. That Write-Host command is therefore added to the function.

```

function Add-Apps(){
 #Add apps

 Write-Host "Creating apps ..." -ForegroundColor Yellow

 Create-MyApp -ListName "Photos" -AppType PictureLibrary
}

```



```
Create-MyApp -ListName "Tasks" -AppType Tasks
}
```

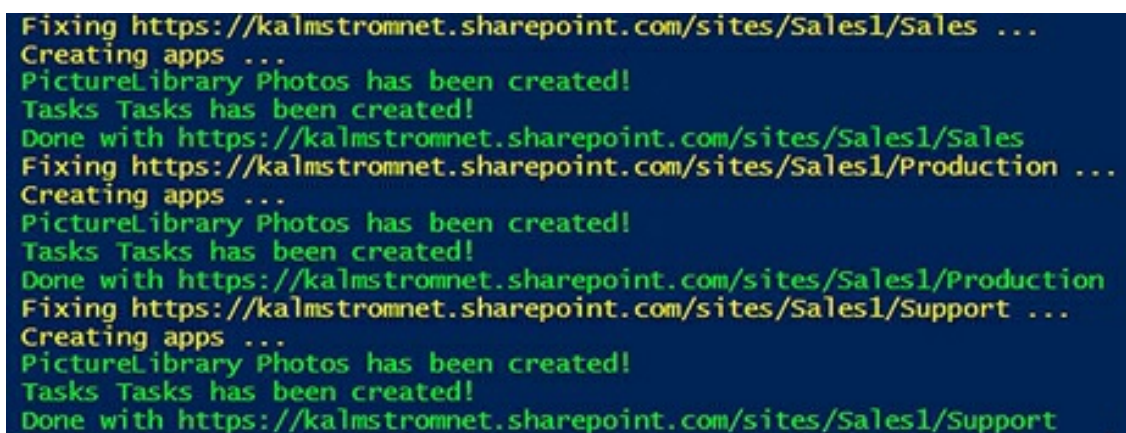
Each app name is already specified in a Write-Host command that we added to the Create-My-App function in an earlier chapter, but we must convert the \$AppType value to a string before we can use it in the progress message.

```
Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
}
```

The subsite navigation and pages are created after the apps, with calls to other functions, and we can of course add yellow "Creating navigation" and "Creating pages" messages in these functions too. I have decided to not do that here. Instead, we will just give a message when the work with each subsite is finished.

```
Write-Host ("Done with " + $CurrWebURL) -ForegroundColor Green
}
```

The image below shows all kinds of messages from the work with the subsites.



```
Fixing https://kalmstromnet.sharepoint.com/sites/Sales1/Sales ...
Creating apps ...
PictureLibrary Photos has been created!
Tasks Tasks has been created!
Done with https://kalmstromnet.sharepoint.com/sites/Sales1/Sales ...
Fixing https://kalmstromnet.sharepoint.com/sites/Sales1/Production ...
Creating apps ...
PictureLibrary Photos has been created!
Tasks Tasks has been created!
Done with https://kalmstromnet.sharepoint.com/sites/Sales1/Production ...
Fixing https://kalmstromnet.sharepoint.com/sites/Sales1/Support ...
Creating apps ...
PictureLibrary Photos has been created!
Tasks Tasks has been created!
Done with https://kalmstromnet.sharepoint.com/sites/Sales1/Support ...
```

Finally, we will also add two Write-Host commands to the Remove-All function.

```
function Remove-All(){
```

```

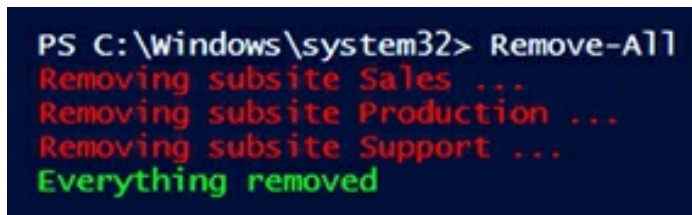
foreach($Dep in $Departments){
 Write-Host ("Removing subsite " + $Dep + " ... ") -ForegroundColor Red

Remove-PnPWeb -Url $Dep -Force
}
$AllNodes = Get-PnPNavigationNode -Location QuickLaunch

foreach($Node in $AllNodes){
 # $Node = $AllNodes[0]

if($Node.Title -eq "kalmstrom.com"){
 Remove-PnPNavigationNode -Identity $Node.Id -Force
}
}
Write-Host ("Everything removed") -ForegroundColor Green
}

```



```

PS C:\Windows\system32> Remove-All
Removing subsite Sales ...
Removing subsite Production ...
Removing subsite Support ...
Everything removed

```

## 27.2 STEPS

We will add Write-Host commands in the foreach loop that creates subsites, in the foreach loop that adds navigation, apps and pages to the subsites, in the Add-Apps function and in the Remove-All function.

When that is done, we will run the whole script, so make sure that you have removed everything from the previous chapters.

1. Enter Write-Host after the start curly brackets in the foreach loop that creates subsites.
2. Start the value with a string that states that the subsite is being created.
3. Add the current department and a string with three dots.
4. Add the parameter ForegroundColor with the value Yellow.

```

Write-Host ("Creating subsite " + $Dep + " ...") -ForegroundColor Yellow

```

5. Enter Write-Host before the end curly bracket of the same loop.
6. Add a value that states that the subsite has been created for the current department. Make the ForegroundColor value Green.

```
Write-Host ("Done with subsite " + $Dep) -ForegroundColor Green
```

7. Enter a Write-Host cmdlet in the foreach loop that works with the subsites, above the Connect-PnPOnline command row.
8. Add the string Fixing and the variable for the URL of the current subsite followed by three dots. Make the color yellow.

```
Write-Host ("Fixing " + $CurrWebURL + " ... ") -ForegroundColor Yellow
```

9. Enter Write-Host before the end curly bracket in the loop that works with the subsites.
10. Add a value that states in green that the loop is done with the URL of the current subsite.

```
Write-Host ("Done with " + $CurrWebURL) -ForegroundColor Green
```

11. Enter Write-Host after the start curly bracket in the Add-Apps function.
12. Add a value that states in yellow that the loop is creating apps.

```
Write-Host "Creating apps ... " -ForegroundColor Yellow
```

13. In the Remove-All function, enter Write-Host after the start curly brackets.
14. Add a value that states in red that the current subsite is being removed.

```
Write-Host ("Removing subsite " + $Dep + " ... ") -ForegroundColor Red
```

15. Before the last end curly bracket of the Remove-All function, enter Write-Host and the value Everything removed in green.

```
Write-Host ("Everything removed") -ForegroundColor Green
```

16. Run the whole script and check that the progress messages are shown and that everything is created in SharePoint.

Note that nothing will be removed in the process, even though the

Remove-All function is included in the run – because there is no call to that function in the script.

Demo:

<https://www.kalmstrom.com/Tips/PowerShell-SharePoint/994-Small-Bussiness-Intranet-5-Progress-PowerShell-SharePoint.htm>

## **27.3 SUMMARY**

This was again a chapter where we modified the script without changing anything in SharePoint. Instead, we added a lot of progress messages, so that you can see more clearly in the console pane what happens when the script is running.

In the next chapter, we will give each subsite its own theme.

## ***28 CREATE A SHAREPOINT INTRANET – THEMES***

In this chapter, we will give each of the subsites its own, custom theme. We will not spend much time with the actual theme creation here, as this is not what I want to show. I will just give each department its own color. Each theme will have the same name as the subsite that uses it.

To be able to add these custom themes to the subsites, we must first add them to the tenant. Therefore, you need to be a SharePoint administrator to perform this exercise. Otherwise, the cmdlet we are using to add the themes to the tenant will not work. (If you already have themes that you want to use in the tenant, you can use the steps from point 30 to add these themes to the subsites.)

### **28.1 THEORY**

We need to perform multiple actions to add custom themes to the subsites:

- To create the themes we will use Microsoft's UI Fabric Theme Designer at <https://fabricweb.z5.web.core.windows.net/pr-deploy-site/refs/heads/master/theming-designer/index.html>. We will create three different themes in this tool.
- We will create a new function, Add-Themes. When we call that function, the themes will be added to the tenant.
- The Add-Themes function will have three variables, one for each department theme. The variable values are hashtables that contain the theme code generated in the UI Fabric Theme Designer. Here the hashtable stores properties, not key-value pairs as in earlier exercises.
- The Add-Themes function will also have three commands with the cmdlet Add-PnPtenantTheme. This cmdlet has three mandatory parameters: Identity (the name), Palette (the actual theme) and IsInverted.

The IsInverted parameter can have two values, \$true and \$false. These are

built-in PowerShell variables and do not have to be declared. PowerShell has many such automatic variables, *refer to* [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_automatic\\_variables?view=powershell-7](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view=powershell-7)

- We will add the themes to the subsites by adding a command with the Set-PnPWebTheme cmdlet to the foreach loop that connects to the subsites. (It is of course possible to add this command to a separate function too, and just call it from the foreach loop, but this command is so short that it would not be of much value.)
- To remove the themes from the tenant, we will add a piece of script to the Remove-All function.

## 28.2 STEPS

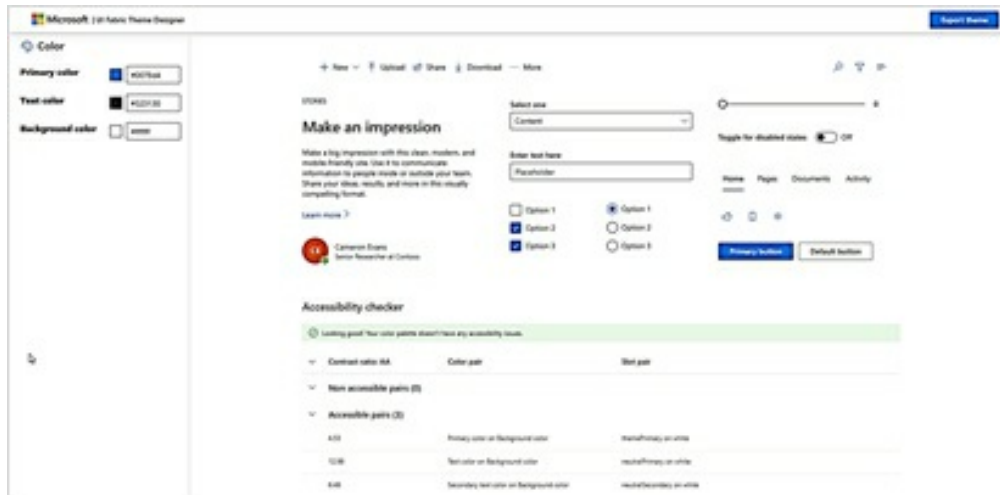
In the previous chapter we did not remove the intranet, so I will assume that you still have it.

### 28.2.1 Function for Tenant Themes

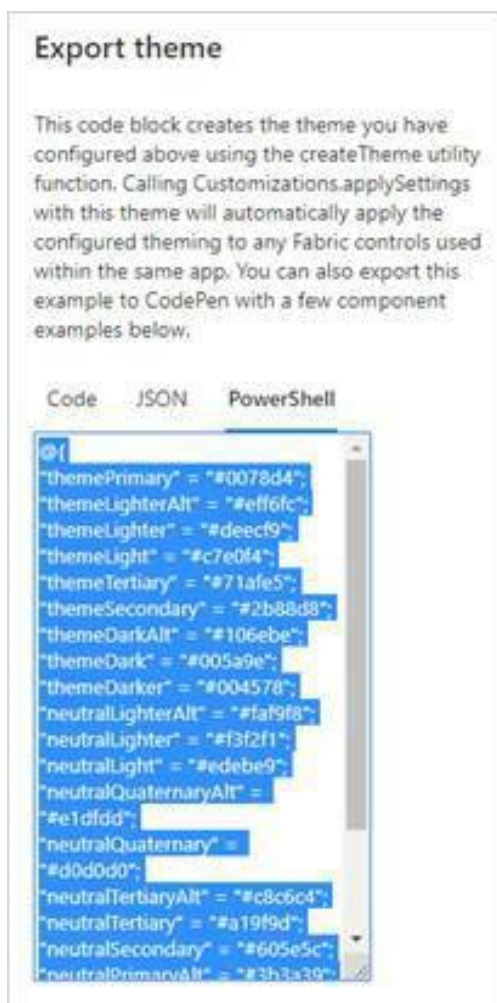
Here we will add a function that adds the themes to the tenant. It will be called from the command prompt.

There are many steps below, but several of them are very similar so this exercise is not as complicated as it might look at a first glance!

1. Open the UI Fabric Theme Designer and create a theme for the Sales subsite.



2. Click on the Export theme button in the top right corner.
3. Select the PowerShell option, select the code with Ctrl+A and copy it.



4. Go to PowerShell ISE and the intranet script we have been working with

earlier

5. Start adding a function above the first foreach loop, that creates the subsites. Name it Add-Themes. It should not have any input parameters.
6. After the start curly bracket, add the variable SalesTheme. Its value should be a hashtable.
7. Paste the theme code you copied from the UI Fabric Theme Designer within the hashtable's curly brackets. (Below I have just added the first part of the theme code, as it is long and not interesting for the understanding of this exercise.)

```
function Add-Themes(){
 $SalesTheme = @{
 "themePrimary" = "#7bd400";
 "themeLighterAlt" = "#f9fdf3";
 "themeLighter" = "#e8f8d0";
 "themeLight" = "#d4f2a9";
 "themeTertiary" = "#ace55c";
 "themeSecondary" = "#89d91a"; ...
 }
}
```

8. Collapse the SalesTheme variable.
9. Enter the cmdlet Add-PnPTenantTheme.
10. Add the parameter Identity and the value Sales.
11. Add the parameter Palette. The value should be the SalesTheme variable.
12. Add the parameter IsInverted with the value \$false.

```
function Add-Themes(){
 $SalesTheme = @{...}
 Add-PnPTenantTheme -Identity "Sales" -Palette $SalesTheme -IsInverted $false
```

13. Go back to the UI Fabric Theme Designer and create a theme for the



Support site.

14. Click on Export theme, select the PowerShell option and copy the theme code.
15. Go back to PowerShell ISE and add another variable to the Add-Themes function: SupportTheme, with the value of a hashtable.
16. Paste the theme code within the hashtable's curly brackets and collapse the SupportTheme variable.
17. Enter the cmdlet Add-PnPTenantTheme.
18. Add the parameters Identity, with the value Support, Palette, with the value of the SupportTheme variable, and IsInverted, with the value \$false.
19. Go back to the UI Fabric Theme Designer and create a theme for the Production site.
20. Click on Export theme, select the PowerShell option and copy the theme code.
21. In PowerShell ISE, add another variable to the Add-Themes function: ProductionTheme, with the value of a hashtable.
22. Paste the theme code within the hashtable's curly brackets and collapse the ProductionTheme variable.
23. Enter the cmdlet Add-PnPTenantTheme.
24. Add the parameters Identity with the value Production, Palette with the value of the ProductionTheme variable and IsInverted with the value \$false.

Enter a progress message that gives the text Themes added in green color.

25. Now the whole function looks like this: (I am showing the function with collapsed hash tables.)

```
function Add-Themes(){
 $SalesTheme = @{...}
```

```
Add-PnPTenantTheme -Identity "Sales" -Palette $SalesTheme -IsInverted $false
```

```
$SupportTheme = @{...}
```

```
Add-PnPTenantTheme -Identity "Support" -Palette $SupportTheme -IsInverted $false
```

```
$ProductionTheme = @{...}
```

```
Add-PnPTenantTheme -Identity "Production" -Palette $ProductionTheme -IsInverted $false
```

```
Write-Host "Themes added" -ForegroundColor Green
```

```
}
```

26. Initialize the function.

27. Call the function to add the themes to the tenant.

```
Add-Themes
```

28. Check that the themes are added to the tenant by entering the cmdlet Get-PnPTenantTheme at the command prompt. That will give you all the tenant themes with their names and the first part of the palette.



| Name       | Palette                                                                                             |
|------------|-----------------------------------------------------------------------------------------------------|
| Sales      | [[themeLight, #d4f2a9], [themeTertiary, #ace55c], [black, #494847], [neutralSecondary, #a3a2a0]...] |
| Support    | [[themeLight, #f2b5a9], [themeTertiary, #e5725c], [black, #494847], [neutralSecondary, #a3a2a0]...] |
| Production | [[themeLight, #f2b5a9], [themeTertiary, #e5b05c], [black, #494847], [neutralSecondary, #a3a2a0]...] |

## 28.2.2 Add Themes to Subsites

Now we will add a command in the second foreach loop that adds the themes to the subsites. We could of course create a function for the themes also and just add another call in the foreach loop. However, this command is so short that a function would be of little or no value.

1. Enter the cmdlet Set-PnPWebTheme below the function calls in the foreach loop that connects to the subsites.
2. Add the default parameter Theme and give it the value of the current subsite (as the themes have the same names as the subsites).

```
Set-PnPWebTheme -Theme $Dep
```

3. Comment out the function calls, because we did not remove anything in the last chapter, so the navigation and the apps and pages are already in place.
4. Run the foreach loop and check that the themes have been added to the subsites.

### 28.2.3 Remove Themes

The last step is to add another command in the Remove-All function that will removes the themes.

1. In the Remove-All function, enter the cmdlet Remove-PnPTenantTheme before the end curly bracket in the first foreach loop.
2. Add the parameter Identity with the value of the current department.
3. Now the whole Remove-All function looks like this:

```
function Remove-All(){
 Connect-PnPOnline $URL

 foreach($Dep in $Departments){
 Write-Host ("Removing subsite " + $Dep + " ... ") -ForegroundColor Red

 Remove-PnPWeb -Url $Dep -Force

 Remove-PnPTenantTheme -Identity $Dep
 }
 $AllNodes = Get-PnPNavigationNode -Location QuickLaunch

 foreach($Node in $AllNodes){
 if($Node.Title -eq "kalmstrom.com"){
 Remove-PnPNavigationNode -Identity $Node.Id -Force
 }
 }
}
```

```
}
Write-Host ("Everything removed") -ForegroundColor Green
}
```

4. Call the Remove-All function at the command prompt.
5. Check that the progress messages we added to the Remove-All function in the previous chapter are displayed as they should and that everything is removed from SharePoint.

## 28.3 SUMMARY

Now you have created your first function without cut and paste! This function adds themes to the tenant with Add-PnPTenantTheme commands.

We added the cmdlet Set-PnPWebTheme and the Theme values in the foreach loop, to add the themes to the subsites. We also added a Remove-PnPTenantTheme command in the Remove-All function.

In both these commands, we could use the name of the current department as the value, because we specified the themes Identity parameter to the same as the department name.

In this book's last chapter, we will add a YouTube video to the homepage of the Sales subsite.

## 29 CREATE A SHAREPOINT INTRANET – ADD VIDEO

In this last chapter, we will add a YouTube video the homepage of the Sales subsite. It is a modern page, so we will contain the video in the modern Content Embed web part.

We can get the embed code from YouTube, but adding it into PowerShell requires some steps. However, if you are going to add many videos to many sites, it will be worth the extra effort! You only need to replace the URL and the video code in the script.

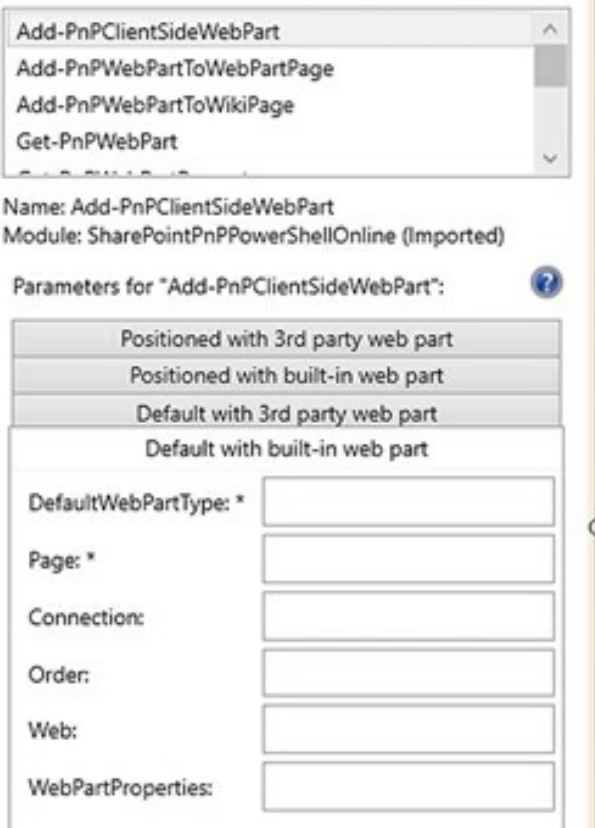
I am using the introduction video for the kalmstrom.com product *Kanban Task Manager* as an example here.



### 29.1 THEORY

In this exercise, we will use the cmdlet `Add-PnPClientSideWebPart`, which adds a client-side (modern) web part to an existing client-side page. We will use the cmdlet with the alternative `Default` with built-in web part.

This alternative requires the parameters `DefaultWebPartType` and `Page`. We will also use the `WebPartProperties` parameter for the actual video code.



The settings of the modern SharePoint Content Embed web part, are stored and set via a Json property on the web part. Therefore, we must build a Json string from the YouTube embed code to use it in PowerShell. This is what we will work with during the major part of this exercise, before we finally can add the Json string to the `Add-PnPClientSideWebPart` cmdlet.

```
Add-PnPClientSideWebPart -Page "Home" -DefaultWebPartType ContentEmbed -WebPartProperties $WebpartPropsJson
```

We will use a class when we build the Json string, so you might want to look back on the Theory section in chapter 19, where I introduce the class concept.

## 29.2 STEPS

Before you start adding the video, you should run the whole script to once more create the subsites, the common navigation, the apps, pages and themes that we removed in the previous chapter.

Start at the command prompt with a call to the `Add-Themes` function, as we

don't have that call in the script. When the themes have been added to the tenant, you can run the script and create everything again.

When everything is in place, it is time to add the video.

1. In YouTube, find a demo that you want to embed, click on Share and select the Embed option.

2. Modify the code if you wish and copy it.

3. In PowerShell ISE, at the end of the script but above the Remove-All function, add a command that connects to the Sales subsite.

```
Connect-PnPOnline ($URL + "/Sales/")
```

(In this case, the value of the Url parameter can be hardcoded also)

4. Create a new variable, YouTubeVideoEmbedCode, and paste the embed code in a string as its value. Use single quotation marks, because the embed code contains double quotes.

5. `Connect-PnPOnline ($URL + "/Sales/")`

```
$YouTubeVideoEmbedCode = '<iframe width="560" height="315"
src="https://www.youtube.com/embed/87pKU-YAVRs" frameborder="0" allow="accelerometer;
autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>'
```

6. Create a new class named WebPartProperties.

7. Add the property embedCode, which is a built-in value for the WebPartProperties parameter, to the class and type it a string.

```
Class WebPartProperties{
 [string]$embedCode
}
```

8. Add a new object, the variable YTVideoProps.

9. For the value, add the New-Object cmdlet with the value WebPartProperties, to create an instance of the class.

```
$YTVideoProps = New-Object WebPartProperties
```

10. Run the class and the YTVideoProps row to get IntelliSense for the

variable.

11. Enter the YTVideoProps variable again and select the EmbedCode property. The value should be the YouTubeVideoEmbedCode variable.

```
$YTVideoProps.embedCode = $YouTubeVideoEmbedCode
```

12. Enter another string variable: WebpartPropsJson.

13. Set its value to the calculation of the ConvertTo-Json cmdlet and the value of the variable YTVideoProps. This will convert the object YTVideoProps to a Json string.

```
[string] $WebpartPropsJson = (ConvertTo-Json $YTVideoProps)
```

14. Run the row with the WebpartPropsJson variable and check that the console pane displays the embed code.

15. Enter the cmdlet Add-PnPClientSideWebPart.

16. Add the parameters Page with the value Home, DefaultWebPartType with the value ContentEmbed and WebPartProperties with the value of the WebpartPropsJson variable.

17. Now everything we have built in this exercise looks like this, with my comments:

```
Connect-PnPOnline ($URL + "/Sales/")
```

```
#Building json string
```

```
#Change the value here for another video
```

```
$YouTubeVideoEmbedCode = '<iframe width="560" height="315"
```

```
src="https://www.youtube.com/embed/87pKU-YAVRs" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>'
```

```
Class WebPartProperties{
```

```
 [string]$embedCode
```

```
}
```



```
$YTVideoProps = New-Object WebPartProperties

$YTVideoProps.embedCode = $YouTubeVideoEmbedCode

[string] $WebpartPropsJson = (ConvertTo-Json $YTVideoProps)
#End building json string

Add-PnPClientSideWebPart -Page "Home" -DefaultWebPartType ContentEmbed -
WebPartProperties $WebpartPropsJson
```

18. Run the Add-PnPClientSideWebPart command and check that the video has been added to the Sales homepage.

## 29.3 SUMMARY

In this last exercise we added embed code from YouTube as a value to the variable YouTubeVideoEmbedCode. We created a new object from a class with the embedCode property and gave it the value of the variable.

Another variable, WebpartPropsJson, was given the value of the object converted to a Json string, and then we could finally add that variable as a value to the WebPartProperties parameter in a Add-PnPClientSideWebPart command.

Many steps for just adding one video, but I hope you learned some coding from it, and once you have the script it is easy to reuse.

Thank you for following me all the way!

# *TO CONTINUE*

If time permits, I want to write a continuation of this book and take up more advanced coding, but now you should also be able to benefit from information online. Here I have gathered some facts that I hope will be useful to you in your further exploration of the PowerShell possibilities.

## **Terms**

When you search for information online, you will probably stumble upon expressions that are not mentioned in this book and that you don't understand. These are possibly some of them:

- **Alias:** Cmdlets, functions and other PowerShell executables can have shorter names, called aliases. There are built-in aliases, and you can also create your own.
- **Argument:** the same as parameter. I have used the word parameter in this book.
- **Expression:** something in the code that PowerShell can determine a value for. Another way to put it, is that expressions are anything in the code that PowerShell uses to return values.

This means that the word can refer to a lot of different things, so I have tried to be more specific this book. That way, I hope you have learned the correct words for different parts of the code and can use and understand them in your further exploration of PowerShell.

- **Pipeline:** The **pipe** operator, |, is used to connect commands, so that the output of each command is used as input to the next command. Such a row of commands is called a pipeline.

I have not used pipelines in this book. I find it clearer to write each command on a new row, instead of having one long pipeline on the screen, that requires horizontal scrolling. You can for example use a pipeline instead of a foreach loop.

## Websites

Here are some websites with PowerShell info that I have found useful:

- At <https://docs.microsoft.com/powershell/module/sharepoint-pnp/?view=sharepoint-ps>, Microsoft gives information about cmdlets that can be used with SharePoint.
- <https://www.powershellgallery.com> is a central repository for sharing and acquiring PowerShell code.
- <https://github.com/SharePoint/PnP-PowerShell> is the open source hub for the SharePoint module used in this book.
- At <https://docs.microsoft.com/en-us/powershell/sharepoint> you will find more information on how to manage PowerShell with SharePoint cmdlets.

## Tool

Finally, I want to mention my free *PnP List Generator*, a PowerShell tool for SharePoint list creation and data import. Now that you have basic knowledge of PowerShell, you will hopefully understand the tool and find it useful for building more advanced SharePoint apps than the ones we have created in this book.

You can use the *PnP List Generator* on the kalmstrom.com website:

<https://www.kalmstrom.com/Tools/PnP-Generator.html>

To create a SharePoint list, enter your preferred content type and site columns, and the *PnP List Generator* will build a PowerShell script for the list. Copy the PowerShell code, paste it in Windows PowerShell ISE and run it, and the new list will be created.

To import data to a new SharePoint list, add all data together with values for the new list's content type and site columns to the *PnP List Generator*. A PowerShell script that you can copy and use in PowerShell ISE will be built in this case also.

# SCRIPTS

Here you can find all scripts created in this book. As we have added code to the script continuously, I have used a bigger font for the new code in each chapter and also added some comments.

Code that was not used in the chapter is not included – but might come back in a later chapter, where it is used again. To save space, I have also, after the two first scripts, removed the code that should always start each script:

```
#First time tasks
```

```
#Install-Module -Name SharePointPnPPowerShellOnline
```

```
#$Creds = Get-Credential
```

```
#Add-PnPStoredCredential -Name $URL -Username $Creds.UserName -Password $Creds.Password
```

```
Import-Module -Name SharePointPnPPowerShellOnline
```

```
$URL = "https://kdemo.sharepoint.com"
```

```
Connect-PnPOnline $URL
```

Use these scripts to check your understanding! When you have gone through the whole book, I hope you will know what each piece of code is doing there.

## 3

The execution policy was set already in chapter 2, and in chapter 3 we installed the SharePoint PnPPowerShellOnline module and imported it.

After that, we connected to a SharePoint site and checked the connection by getting the apps in that site.

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted
```

```
Install-Module -Name SharePointPnPPowerShellOnline
```

```
Import-Module -Name SharePointPnPPowerShellOnline
```

```
Connect-PnPOnline "https://kdemo.sharepoint.com"
```

## Get-PnP-List

### 4

In chapter 4, I showed two ways of saving the SharePoint credentials in the PowerShell script, and we also saved the URL to the site in a variable.

With Web login:

```
#First time tasks
#Install-Module SharePointPnPPowerShellOnline
Import-Module -Name SharePointPnPPowerShellOnline
$URL = "https://kdemo.sharepoint.com"
Connect-PnPOnline $URL -UseWebLogin

Import-Module -Name SharePointPnPPowerShellOnline
Connect-PnPOnline $URL
Get-PnPList
```

With Windows Credential Manager:

```
#First time tasks
#Install-Module SharePointPnPPowerShellOnline
$URL = "https://kdemo.sharepoint.com"
#Store Credential in Windows Control Panel Credential Manager

$Creds = Get-Credential

Add-PnPStoredCredential -Name $URL -Username $Creds.UserName -
Password $Creds.Password

#End first time tasks

Import-Module -Name SharePointPnPPowerShellOnline
Connect-PnPOnline $URL
```

Get-PnPList

## 5

In chapter 5, we created a SharePoint generic list app. It has versioning enabled and a link added to the Quick launch, and we made sure that we get an app URL without spaces.

We also created a command that removes the app.

```
New-PnPList -Title "Hello World List" -Template GenericList -
EnableVersioning -OnQuickLaunch -Url "HelloWorld"
```

```
Remove-PnPList "Hello World" -Force
```

## 6

In chapter 6, we built a function for the generic list app creation and added the New-PnPList command that we created in chapter 5 to it. We also added a Write-Host command to the function.

After that, we called the function to create a new app. We used the function's input parameters in the call and gave them string values.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ("List " + $ListName + " has been created!") -
ForegroundColor Green
}
```

```
Create-MyList -ListName "Hello World" -ListURL "HelloWorld"
```

```
Remove-PnPList "Hello World" -Force
```

## 7

In chapter 7, we built a for-loop that calls the function we built in chapter 6 and creates ten list apps with different names. This for-loop increases the value of an integer variable with one for each loop, so that each app gets a different number added to the name.

We also built another for-loop to remove the ten lists.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}

for($i=1;$i -lt 11;$i++){
 Create-MyList -ListName ("Hello World" + $i) -ListURL ("HelloWorld" +
 $i)
}

for($i=1;$i -lt 11;$i++){
 Remove-PnPList ("Hello World" + $i) -Force
}
```

## 8

In chapter 8, we used Excel to create multiple calls to the function we built earlier. Removal commands were also created in Excel.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

```
Create-MyList -ListName HR -ListURL HR
```

```
Create-MyList -ListName Marketing -ListURL Marketing
```

```
Create-MyList -ListName IT -ListURL IT
```

```
Create-MyList -ListName Sales -ListURL Sales
```

```
Create-MyList -ListName Engineering -ListURL Engineering
```

```
Remove-PnPList HR
```

```
Remove-PnPList Marketing
```

```
Remove-PnPList IT
```

```
Remove-PnPList Sales
```

```
Remove-PnPList Engineering
```

## 9

We used Excel in chapter 9 too, but here we created an array with department names. The array is used in a foreach loop that runs through the function we built earlier and creates apps with the same names as the departments.

The same array is used in another foreach loop that removes the department apps.

```
function Create-MyList($ListName, $ListURL){
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}

$Departments = 'HR','Marketing','IT','Sales','Engineering'

foreach($Dep in $Departments){
 Create-MyList -ListName $Dep -ListURL $Dep
}
```



```
foreach($Dep in $Departments){
 Remove-PnPList $Dep -Force
}
```

## 10

In chapter 10 we enhanced the function we created in chapter 6 and then have used in several chapters.

We typed the input parameters and created an If statement that sets the URL to be in lower case letters and without spaces if the ListURL variable has not already been given a value.

To test the enhanced function, we called it with a ListName parameter value that had both capitals and a space.

```
function Create-MyList([string] $ListName, [string] $ListURL){
 if($ListURL -eq ""){
 $ListURL = $ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template GenericList -EnableVersioning -OnQuickLaunch -Url
 $ListURL

 Write-Host ("List " + $ListName + " has been created!") -ForegroundColor Green
}
```

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

```
foreach($Dep in $Departments){
 Create-MyList -ListName $Dep
}
```

```
Create-MyList "Hello World"
```

```
foreach($Dep in $Departments){
 Remove-PnPList $Dep -Force
}
```

## 11

In chapter 11 we used the function that creates a generic list app as a template and built a function that creates a SharePoint document library.

We also used the foreach loop with the call to the list app function as a template for a foreach loop that calls the new library function.

```
function Create-MyLibrary([string] $ListName, [string] $ListURL){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }

 New-PnPList -Title $ListName -Template DocumentLibrary -Url
 $ListURL

 Write-Host ("Library " + $ListName + " has been created!") -ForegroundColor Green
}

foreach($Dep in $Departments){
 Create-MyLibrary -ListName $Dep
}

foreach($Dep in $Departments){
 Remove-PnPList $Dep -Force
}
```

## 12

In chapter 12, we used the earlier functions as templates to build a function that can be used for all app types. As this function only has a general app type, we needed to enter a specific app type parameter in the foreach loop that calls the function.

We also changed the functions for creation of generic lists and libraries that we built earlier, so that they now call the general app function.

```
function Create-MyList([string] $ListName, [string] $ListURL){
```

```

 Create-MyApp -ListName $ListName -ListURL $ListURL -AppType
GenericList
}

function Create-MyLibrary([string] $ListName, [string] $ListURL){
 Create-MyApp -ListName $ListName -ListURL $ListURL -AppType
DocumentLibrary
}

function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template $AppType -EnableVersioning -OnQuickLaunch -Url
$ListURL

Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
}

$Departments = 'HR','Marketing','IT','Sales','Engineering'

foreach($Dep in $Departments){
 Create-MyApp -ListName $Dep -AppType Tasks
}

foreach($Dep in $Departments){
 Remove-PnPList $Dep -Force
}

```

## 13

In chapter 13 we left app creation and built a foreach loop that runs through the same departments array as we used earlier and creates sites collections.

We also created a foreach loop that runs through the array and removes the site collections again.

```
$Departments = 'HR','Marketing','IT','Sales','Engineering'
```

```
foreach($Dep in $Departments){
```

```
 $SiteURL = "https://kdemo.sharepoint.com/sites/" + $Dep
```

```
Write-Host ($SiteURL + " is being created ...") -ForegroundColor Yellow
```

```
New-PnPSite -Type CommunicationSite -Title $Dep -Url $SiteURL
```

```
}
```

```
foreach($Dep in $Departments){
```

```
 $SiteURL = "https://kdemo.sharepoint.com/sites/" + $Dep
```

```
Write-Host ($SiteURL + " is being removed ...") -ForegroundColor Red
```

```
Remove-PnPtenantSite -Url $SiteURL -Force -SkipRecycleBin
```

```
}
```

## 14

In chapter 14, we built a function that imports data from an Excel sheet with one column to a SharePoint list app. After that, we created calls to the function in Excel. We used a hashtable with a key-value pair to specify where the data from Excel should be entered in SharePoint.

```
function Create-MyApp([string] $ListName, [string] $ListURL,
```

```

[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template $AppType -EnableVersioning -OnQuickLaunch -Url
 $ListURL

Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
}

Create-MyApp -ListName Countries -AppType GenericList

function Add-Country([string] $CountryName){
 Add-PnPListItem -List "Countries" -Values @{"Title"=$CountryName}
}

Add-Country Argentina
Add-Country Austria
Add-Country Belgium
Add-Country Brazil
Add-Country Canada
Add-Country Denmark
Add-Country Finland
Add-Country France

```

## 15

In chapter 15, we imported data from an Excel sheet with multiple columns to a SharePoint list app. We created an array of the Excel file content and used it in a foreach loop where key-value pairs connected the Excel and SharePoint columns.

```

function Create-MyApp([string] $ListName, [string] $ListURL,
[Microsoft.SharePoint.Client.ListTemplateType] $AppType){
 if($ListURL -eq ""){
 $ListURL =$ListName.Replace(" ", "").ToLower()
 }
 New-PnPList -Title $ListName -Template $AppType -EnableVersioning -OnQuickLaunch -Url
$ListURL

Write-Host ($AppType.ToString() + " " + $ListName + " has been created!") -ForegroundColor
Green
}

Create-MyApp -ListName Customers -AppType Contacts

$Text = Get-Content "C:\Users\PeterKalmström\Documents\Customers.txt"
$Rows = ConvertFrom-Csv -Delimiter "`t" -InputObject $Text
#$Row = $Rows[0]

foreach($Row in $Rows){
 $RowValues =@{}
 $RowValues +=@{'Title' = $Row.'Contact last name'}
 $RowValues +=@{'Company' = $Row.CustomerName}
 $RowValues +=@{'FirstName' = $Row.'Contact first name'}
 $RowValues +=@{'Email' = $Row.'E-mail'}
 Add-PnPListItem -List "Customers" -Values $RowValues
}

```

## 16

In chapter 16 we uploaded files from a local folder to SharePoint. We added

the files to an array and then created a foreach loop that runs through the array and adds the files to a SharePoint library.

```
$Files = Get-ChildItem "C:\Users\PeterKalmström\Documents\ToImport"
```

```
foreach($File in $Files){
```

```
 #$File = $Files[0]
```

```
 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
```

```
}
```

## 17

In chapter 17, we expanded the foreach loop that we created in chapter 16, so that it also moves the uploaded files to a subfolder. In If statement makes sure that the subfolder is excluded from the upload.

The Theory section of chapter 17 also included information about error management.

```
$ErrorActionPreference = "Stop"
```

```
$PSDefaultParameterValues['*:ErrorAction']='Stop'
```

```
$Files = Get-ChildItem "C:\Users\PeterKalmström\Documents\ToImport"
```

```
foreach($File in $Files){
```

```
 #$File = $Files[0]
```

```
 if($File.PSIsContainer -ne $true){
```

```
 Add-PnPFile -Folder "Shared Documents" -Path $File.FullName
```

```
 Move-Item -Path $File.FullName -Destination
```

```
"C:\Users\PeterKalmström\Documents\ToImport\Imported"
```

```
 }
```

```
}
```

## 18

Chapter 18 explains how to run a script automatically on a schedule and has no new code.

## 19

Chapter 19 is a preparation for the process in chapter 21, where we upload files with metadata to SharePoint. In chapter 19 we created the document library and added four custom columns to it, all with simple commands.

```
New-PnPList -Title "CVs" -Template DocumentLibrary
```

```
Add-PnPField -List "CVs" -DisplayName "Decision" -InternalName
"CVDecision" -Type Choice -Choices "Yes","No","Maybe" -
AddToDefaultView
```

```
Add-PnPField -List "CVs" -DisplayName "First Name" -InternalName
"CVFirstName" -Type Text -AddToDefaultView
```

```
Add-PnPField -List "CVs" -DisplayName "Last Name" -InternalName
"CVLastName" -Type Text -AddToDefaultView
```

```
Add-PnPField -List "CVs" -DisplayName "Department" -InternalName
"CVDepartment" -Type Text -AddToDefaultView
```

## 20

Chapter 20 is also a preparation for chapter 21. Here we created an array of all the files and built a class for the metadata properties.

We also built a foreach loop that creates one instance of the class for each file and runs through the array to find the correct metadata for each column in the library.

The foreach loop also creates progress messages, so that we can test the code and see in the console pane if the metadata is correctly structured.



```
Class CV{
 $FirstName

 $LastName

 $Department

 $Decision
}

$AllCVFiles = Get-ChildItem -Path
"C:\Users\PeterKalmström\Documents\CVs" -Recurse -File
foreach($CVFile in $AllCVFiles){

#$CVFile = $AllCVFiles[0]

$CVObj = New-Object CV

$CVObj.Decision = $CVFile.Directory.Name
 $FileNameParts = $CVFile.BaseName -split ' - '

$CVObj.Department = $FileNameParts[1]

$NameParts = $FileNameParts[0] -split ', '

$CVObj.LastName = $NameParts[0]
```

```
$CVObj.FirstName = $NameParts[1]
```

```
Write-Host (ConvertTo-Csv $CVObj)
```

```
}
```

## 21

In chapter 21, we could finally perform the file upload that we prepared in chapters 19 and 20. To do that, we added some more code to the foreach loop: a hashtable of key-value pairs and a command that uploads the files and reads the metadata distribution from the hashtable.

```
Class CV{
```

```
 $FirstName
```

```
 $LastName
```

```
 $Department
```

```
 $Decision
```

```
}
```

```
$AllCVFiles = Get-ChildItem -Path "C:\Users\PeterKalmström\Documents\CVs" -Recurse -File
```

```
foreach($CVFile in $AllCVFiles){
```

```
 $CVObj = New-Object CV
```

```
$CVObj.Decision = $CVFile.Directory.Name
```

```
 $FileNameParts = $CVFile.BaseName -split '-'
```

```
$CVObj.Department = $FileNameParts[1]
```

```
 $NameParts = $FileNameParts[0] -split ','
```

```
$CVObj.LastName = $NameParts[0]
```

```
$CVObj.FirstName = $NameParts[1]
```

```
#Write-Host (ConvertTo-Csv $CVObj)
```

```
 $CVValues = @{ }
```

```
 $CVValues += @{'Title'=$CVFile.BaseName }
```

```
 $CVValues += @{'CVDecision'=$CVObj.Decision }
```

```
 $CVValues += @{'CVFirstName'=$CVObj.FirstName }
```

```
 $CVValues += @{'CVLastName'=$CVObj.LastName }
```

```
 $CVValues += @{'CVDepartment'=$CVObj.Department }
```

```
 Add-PnPFile -Path $CVFile.FullName -Folder "CVs" -Values $CVValues
```

```
}
```

## 22 – 29

In the seven last chapters we create a site collection with subsites, apps, pages, themes and a video, and we added to the same script in all those chapters. Therefore, I will just add the final script here.

We used an array of three departments for the whole intranet. Each department had a subsite with the same name as the department, and that site had a theme that also used the department name.

A foreach loop that runs the array creates subsites and adds navigation to the root site.

Another foreach loop runs the same array, but it first connects to each subsite. Then this foreach loop creates a common navigation, apps, pages and themes for the subsites by calling functions that we created for these actions.

We added multiple progress messages to be able to see the process while we tested, and we also added all removal code to one function.

Finally, we converted a video embed code to a json string and added it to a page.

```
function Create-MyApp([string] $ListName, [string] $ListURL,
```

```

[Microsoft.SharePoint.Client.ListTemplateType] $AppType){...}
$Departments = "Sales","Production", "Support"
foreach($Dep in $Departments){
 #Create the subsites

Write-Host ("Creating sub site " + $Dep + " ...") -ForegroundColor Yellow

New-PnPWeb -Title $Dep -Url $Dep -InheritNavigation -Template
SITEPAGEPUBLISHING#0

#Add navigation node to the top site

Add-PnPNavigationNode -Location QuickLaunch -Title $Dep -Url ($URL +
"/" + $Dep)
 Write-Host ("Done with site " + $Dep + " ...") -ForegroundColor Green
}

function Add-Themes(){
 #Add themes

$SalesTheme = @{...}
 Add-PnPTenantTheme -Identity "Sales" -Palette $SalesTheme -IsInverted
 $false

$SupportTheme = @{...}
 Add-PnPTenantTheme -Identity "Support" -Palette $SupportTheme -
 IsInverted $false

$ProductionTheme = @{...}
 Add-PnPTenantTheme -Identity "Production" -Palette $ProductionTheme -
 IsInverted $false

Write-Host "Themes added" -ForegroundColor Green
}

function Add-Apps(){
 #Add apps

```

```

Write-Host "Creating apps ... " -ForegroundColor Yellow

Create-MyApp -ListName "Photos" -AppType PictureLibrary

Create-MyApp -ListName "Tasks" -AppType Tasks
}

function Add-SubsiteNavigation(){
 #Add navigation between subsites

 foreach($DepSub in $Departments){
 if($Dep -ne $DepSub){
 Add-PnPNavigationNode -Location QuickLaunch -Title $DepSub -
 Url ($URL + "/" + $DepSub)
 }
 }

 #Add navigation to root site and external site

 Add-PnPNavigationNode -Location QuickLaunch -Title "HQ" -Url $URL

 Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -
 Url "https://kalmstrom.com" -External
}

function Add-Pages(){
 #Add pages

 Add-PnPClientSidePage -Name "Progress"

 Add-PnPClientSidePage -Name "Problems"

 Add-PnPClientSidePage -Name "Plans"
}

foreach($Dep in $Departments){
 $CurrWebURL =($URL + "/" + $Dep)
 #Connect to the department site

```

```
Write-Host ("Fixing " + $CurrWebURL + " ... ") -ForegroundColor Yellow
```

```
Connect-PnPOnline $CurrWebURL
```

```
Add-Apps
```

```
Add-SubsiteNavigation
```

```
Add-Pages
```

```
Set-PnPWebTheme -Theme $Dep
```

```
Write-Host ("Done with " + $CurrWebURL) -ForegroundColor Green
```

```
}
```

```
#Connect back to the root site and add external link on the root site
```

```
Start-Sleep -Seconds 5
```

```
Connect-PnPOnline $URL
```

```
Add-PnPNavigationNode -Location QuickLaunch -Title "kalmstrom.com" -
Url "https://kalmstrom.com" -External
```

```
#Add video on subsite homepage
```

```
Connect-PnPOnline ($URL + "/Sales/")
```

```
#Building json string
```

```
#Change here for another video
```

```
$YouTubeVideoEmbedCode = '<iframe width="560" height="315"
src="https://www.youtube.com/embed/WrVC0iIlmaQ" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-
picture" allowfullscreen></iframe>'
```

```
Class WebPartProperties{
 [string]$embedCode
}
```

```

$YTVideoProps = New-Object WebPartProperties
$YTVideoProps.embedCode = $YouTubeVideoEmbedCode

[string] $WebpartPropsJson = (ConvertTo-Json $YTVideoProps)
#End building json string

Add-PnPClientSideWebPart -Page "Home" -DefaultWebPartType
ContentEmbed -WebPartProperties $WebpartPropsJson

function Remove-All(){
 Connect-PnPOnline $URL

 foreach($Dep in $Departments){
 Write-Host ("Removing subsite " + $Dep + " ... ") -ForegroundColor
 Red

 Remove-PnPWeb -Url $Dep -Force

 Remove-PnPTenantTheme -Identity $Dep
 }

 $AllNodes = Get-PnPNavigationNode -Location QuickLaunch

 foreach($Node in $AllNodes){
 # $Node = $AllNodes[0]

 if($Node.Title -eq "kalmstrom.com"){
 Remove-PnPNavigationNode -Identity $Node.Id -Force
 }
 }
 Write-Host ("Everything removed") -ForegroundColor Green
}

```

# ***ABOUT THE AUTHORS***

Peter Kalmstrom is the CEO and Systems Designer of the family business Kalmstrom Enterprises, well known for the software brand *kalmstrom.com Business Solutions*. Besides developing the *kalmstrom.com* products, Peter also creates custom solutions for customers all over the world. He has 19 Microsoft certifications, among them several for SharePoint, and he is a certified Microsoft Trainer.

Peter began developing his *kalmstrom.com* products around the turn of the millennium, but for a period of five years, after he had created *Skype for Outlook*, he also worked as a Skype product manager. In 2010 he left Skype, and since then he has been concentrating on his own company and on lecturing on advanced IT courses.

Peter has published seven more books: *SharePoint Online from Scratch*, *SharePoint Online Exercises*, *SharePoint Online Essentials*, *Office 365 from Scratch*, *SharePoint Flows from Scratch*, *SharePoint Workflows from Scratch* and *Excel 2016 from Scratch*. All are sold worldwide via Amazon.

As a preparation for lectures and books, Peter has created various video demonstrations, which are available on YouTube and at <http://www.kalmstrom.com/Tips/PowerShell-SharePoint>.

Peter divides his time between Sweden and Spain. He has three grownup children, and apart from his keen interest in development and new technologies, he likes to sing and act. Peter is also a dedicated vegan and animal rights activist.

Kate Kalmström is Peter's mother. She is a former teacher, author of schoolbooks and translator who now works in the family business and assists Peter with his books.