Windows Powershell v3

# Introduction

Windows Server has been struggling with a problem for a while now and this is the Graphical User Interface or GUI. The GUI was created to make a system administrator’s life easier, it provides an easy to use environment in which the system administrator can perform his tasks. All the system administrator has to do is simply click… well a whole bunch of simple clicks.

The GUI is a great tool, it is however lacking in multiple areas. The main problem is clicking, there is way too much of it and it is way too slow. You have to click open your tool, wait for it to load, click a bunch more times to finally get what you want. Another problem is automatisation, a system administrator has to spend way too much time performing his tasks.

When you look at Linux this all performs way better. Here scripts and automatisation were the focus, powerful commands could be passed through a simple command line interface. Microsoft took this concept and created Windows PowerShell, a powerful command line interface where automatisation, scripting and remoting were the focus. Now many Windows system administrators are making the jump from the traditional GUI to the powerful Windows PowerShell.

# Using Powershell, tips and tricks

* 1. **Which powershell do I use?**

There are two different versions of PowerShell:

* Windows PowerShell  
  This is the environment where you can execute powerful, short commands and receive sorted, filtered and formatted results which can be displayed in the console or be redirected to a file format of your liking

.

* Windows PowerShell ISE  
  This is the Integrated Scripting Environment. Here is where you are going to write, test and save your scripts.   
  Windows PowerShell ISE doesn’t necessarily have to be used for scripting, you can also use it to take advantage of coloring, drop-down lists and automatic parameter revelation features. It also has an Show Command Add-on feature in which you can create commands from a graphical environment.  
  The Show-Command is an interesting cmdlet because it offers a visual view of many cmdlets and the respective help.
  1. **What do I have access to?**

There are two ways to launch Windows PowerShell

:

* As administrator  
  This means you’ll have access to everything.
* As user  
  This means your access will be limited to only the features which you’re allowed access to. So you won’t be able to install any drivers or read any security logs. If you were to run a command which requires administrator rights you’ll receive a ‘Permission denied error’.
  1. **Cmdlets**

In Powershell you’ll be working with cmdlets, these are light-weight powershell functions that execute one specific function.

All cmdlets will have the same structure which is: verb-noun  
Some frequent verbs are: get, set, add, show, remove and format. You’ll be most frequently using get.  
Some examples of nouns are: process, service, disk and netadapter.

To make the cmdlets work, you have to combine the verb and the noun with a simple   
dash.  
For instance: Get-Process, Get-Service, Get-Date, …  
Example: Show-Command -Name Get-Process

.

* 1. **Parameters**

To make efficient use of the cmdlets you’ll have to utilize parameters, these are supplied behind a cmdlet. Every cmdlet will have it’s own supply of parameters, in the next chapter we’ll see how to efficiently query these.

Some frequently returning parameters:

* Verbose

→ This will return detailed information to the output

* Debug
* WarningAction
* WarningVariable
* ErrorAction

→ When you want to hide any errors in the PowerShell output

* ErrorVariable
* OutVariable
* OutBuffer
* Whatif
* Confirm
  1. **If I pass this command… oops!**

With a such a powerful tool as powershell comes great responsibility! When it comes to executing powerful commands it’s better to be safe than sorry and see what the command will actually accomplish.  
  
There are some handy parameters just for this:

* -Whatif  
  When you place this parameter at the end of a command it won’t execute the command but simulate what would happen if you would execute it. For every operation the command execute there will be a notification on how this operation affects an object. Afterwards the system administrator can decide if the command is safe for production.  
    
  For example:  
  Get-Service | Stop-Service  
  → This command gathers all the services and commands them to stop, which will have disastrous consequences.  
    
  Get-Service | Stop-Service -Whatif

→ Instead of completing the command PowerShell will now display what changes will happen if this command were to be executed. In the output we will see that the operation Stop-Service will be applied to all the services

* Confirm  
  When you place confirm at the end of your command a prompt will appear for every operation that is executed. You will be required to answer each prompt :

[Y] Yes Allows this operation (default)

[A] Yes to All Allows this and every following operation

[N] No Disallow this operation

[L] No to All Disallow this and following operations

[S] Suspend interrupt the command

[?] Help Help

For example:  
You want to stop some active directory services, but not all of them. Thanks to the -Confirm parameter you can decide one for one whatever you want to stop the service or not.  
Get-Service -DisplayName \*active\* | stop-service -confirm  
→ For each service with active in the display name a prompt will be given whatever you want to apply stop-service or not.

# S.O.S. PowerShell

**3.1 Get-Help**

One of the first commands to run when opening Windows PowerShell for the first time is the Update-Help cmdlet. This is because Windows PowerShell does not ship help files with the product. This does not mean that no help presents itself.  
It does mean that help beyond simple syntax display, requires an additional download.

The modular nature of Windows PowerShell requires additional consideration when updating help. Simply running Update-Help does not update all of the modules loaded on a particular system. In fact, some modules may not support updatable help at all. These generate an error when you attempt to update help. The easiest way to ensure you update all possible help functions is to use both the module parameter and the force switched parameter. The command to update help for all installed modules (that support updatable help) is shown here:

Update-Help -Module \* -Force

To suppress the errors during the command and walk through them afterwards you can use this handy script:

UpdateHelpTrackErrors.ps1

$error.Clear()

Update-Help -Module \* -Force -ea 0

For ($i = 0 ; $i -le $error.Count ; $i ++)

{ "`nerror $i" ; $error[$i].exception }

Help functions have gotten a bad name, they’re usually not that well designed and don’t even offer that much help, but the PowerShell Get-Help function is definitely not one of those.  
It has been designed so you can easily find and learn to use the command you need.

To explore the full functionality of Get-Help you can use:

Get-Help -Full

You will get a big page of help consisting of the following:

* **Name**  
  The name of the cmdlet.
* **Synopsis**  
  A short description of what the cmdlet does.
* **Syntax**  
  A list of possible parameters to use with the function.
* **Description**  
  An elaborate description on what the function does.
* **Related Links**  
  All things that can be related to this function.
* **Remarks**  
  If you want any additional help.
* **Examples**  
  Shows different ways to use the cmdlet, there’s a good chance you’ll find an example you can use.

You can also browse the help online:  
Get-Help Get-Services -Online

Or in a handy window with a search function:

Get-Help Get-Process -ShowWindow

You can also ask for a specific part of the help file:  
Get-Help Get-PSDrive -example

But in general you will get the most benefit of the Get-Help function to get more

information about the modules or cmdlets. With the command

Get-Help -Name *any\_module*

**3.1.2. Save-Help cmdlet**

The Save-Help cmdlet downloads the help for modules and saves it on the local

system. To use this function you’ll need to have the administrator rights. It’s a

possibility to specify an alternate path with the LiteralPath or DestinationPath parameters.  
PS C:\> Save-Help -DestinationPath C:\MyHelpDirectory

**3.1.3 Update-Help cmdlet ?**This cmdlet is responsible to make sure the help is downloaded from the internet and installed on the right place. With the general Update-Help command you update all currently loaded modules directly from the internet. If you want to update only a piece of modules it’s possible to use the parameter -SourcePath  
PS C:\ Update-Help -SourcePath C:\MyHelpDirectory

* 1. **Get-Command**

The Get-Command cmdlet gets details of every command available to you. These commands include cmdlets, modules, functions, workflows, aliases, and executable commands. When using the Get-Command cmdlet, you can obtain a list of all the cmdlets installed on Windows PowerShell.

We can refer to Get-Command using its alias to find this alias use of the command:  
Get-Alias G\*

We can see that the alias for Get-Command is gcm

To create your own alias simply call New-Alias give the desired alias and the desired command:

New-Alias gh Get-Help

Get-Command has many uses but we’ll be using it to find the command we desire. We have two ways of going about this:

* Use the -verb parameters  
  gcm -verb se\*  
  This will return all cmdlets with se as parameter, which will result in a long list.
* Using the -noun parameter  
  Use this when you sort of know which command you are looking for. For example you need a cmdlet for a command that contains bit:

gcm -noun \*bit\*

This will return all the cmdlets with active in their name.

We can also use Get-Command to find the syntax of a command, this way we can find out which parameters suit us best:

gcm -syntax gcm

* 1. **Get-Member**

The Get-Member cmdlet retrieves information about the members of objects. You can use the Get-Member cmdlet to examine the methods and properties of objects. When the Get-Member cmdlet is used with Get-ChildItem on the file system, it returns a listing of all the methods and properties available to work with the DirectoryInfo and FileInfo objects. The main goal of the cmdlet is to return the children of the current items specified by the Path or LiteralPath parameters.

To get all the members of a cmdlets:

PS C:\> Get-ChildItem | Get-Member

Only the methods:

PS C:\> Get-ChildItem | Get-Member -membertype method

Select all files in the root drive C:  
PS C:\> Get-ChildItem -File  
Filter all child items that were not directories and hidden, and also the objects that were not directories:  
PS C:\> Get-ChildItem -Attribute !Directory+Hidden,!Directory

**! :** The NOT operator (we do not want that particular type)  
**+:** The AND operator (we want both types)  
**,:** The OR operator (Specify filters where it has to match one of the parameters to give a return)

* 1. **Get-Module**A module is a packaged set of cmdlets, functions, alias, types and other components. It’s easy to view all the cmdlets, just by Get-Module.  
     But if you want to get only the installed cmdlets. You’ll have to use the -ListAvailable parameter.  
     PS C:\ Get-Module -ListAvailable  
     In difference to PowerShell v2 did the now implement the auto-discovery cmdlet. This will make sure that if you want to execute a cmdlet or script, but you need to have something previous installed to run it. The auto-discovery will implement this automatically. When in 2.0 we had to import it manually.
  2. **Transcript**

A handy way to keep track of everything you are doing in Windows Powershell is the transcript command.

Start the Windows PowerShell Transcript command via the Windows PowerShell profile. To do this, add the Start-Transcript cmdlet as it appears here (the Start-Transcript cmdlet creates a record of all Windows PowerShell commands, as well as the output from those commands).

Start-Transcript

# Getting in the pipe

* 1. **What is piping and how does it work?**

The pipeline is one of Windows PowerShell’s most important features, it takes the object earlier called upon and executes another cmdlet on it.  
For example:  
PS C:\ Get-NetAdapter | Disable-NetAdapter  
PS C:\ Get-Service -Name Themes | Start-Service

Pipelining is one of the strongest features available in Windows PowerShell.

This feature makes it possible to make really easy powerful commands.

* 1. **Filter, sort and grouping**

When you’re working with Windows-Powershell and use a get cmdlet to get some information. Chances are that you’re about to get flooded with information, that’s why it is important to filter and sort. When it comes to filtering and sorting there is one important rule: first filter and then sort. If you sort first you’ll be putting a lot more stress on the server since it’s gonna sort **all** the available information, after which you’re just gonna take a tiny bit of that.

Filtering and sorting:

Let’s say I want a list of processes sorted of which process is using the most virtual memory:

Get-Process |Sort-Object -Property VirtualMemorySize

A handy tool here is the tab-completion tool. After properties you can hit tab to go through all of the available properties. If you already know what you’re looking for just type in the first letters and hit tab.  
  
Now I would like to only list the ones with a memory greater than 1000MB:

Get-Process | Where-Object VirtualMemorySize -gt 1000MB

A few handy filtering parameters are: -gt (GreaterThan), -lt (LesserThan) and -eq (equals). For more info about comparison operators:  
PS C:\ Get-Help about\_Comparison\_Operators

Now we can combine both

Get-Process | Where-Object vm -gt 1000MB | Sort-Object -Property VM

We can also work with dates, if I want to find what drivers were installed on a specific date:

Get-WindowsDriver -Online | where date -gt 10/8/2012

Or after a specific date:

Get-HotFix | Where installedon -gt 12/1/12 |sort installedon

We can also use select to only display the information we want:

Get-AppxPackage | Select Name, Version, Publisher | Where Publisher -Match Microsoft | Sort Name

Group-Object

We can also separate objects into groups to get an easy overview of the information.  
If we want to see how many services are in which status at the moment:

Get-Service | Sort-Object status | Group-Object -Property status

Or shortened:

Get-Service | sort status | group status

Select-Object

If you want to select multiple properties from an object, and not using a script here for, the Select-Object helps with it. The following command will return a list of the process ID(PID), and the values of each.

PS C:\> Get-Process –Name PowerShell | Select-Object –ExpandProperty ID

ForEach-Object

People who are more into scripting will use the ForEach-Object more than the Select-Object. Although both can give the same return or output, they differ from each other.

PS C:\> Get-Service | **Select-Object** DisplayName,Status  
PS C:\> Get-Service | **ForEach-Object** { @{DisplayName=$\_.

DisplayName;Status=$\_.Status} }

It’s obvious that the Select-Object is easier to write down, but what the ForEach-Object has over the Select-Object is that u can write down anything u want between the script block. For example if u want to call a method instead of a property, you have to use ForEach-Object.

There is also the possibility to work with strings using the ForEach-Object.  
PS C:\> "a Bird!","a Plane!","Superman!" | ForEach-Object Insert -ArgumentList 0,"It's "  
  
**Output:**

It's a Bird!

It's a Plane!

It's Superman!

* 1. **Formatting and storing output**

When you’re looking up information it’s important that you only display the information you need and that you display it in a nicely ordered table.

Formatting output:  
  
We can choose the specific order of the output:

Get-Process | Format-Table -Property name, handles, vm, ws

If you prefer a list:

Get-Process | Format-List -Property \*

To make a tightened up display in the output with multiple columns:

Get-Process | Format-Wide -Property name -AutoSize

Shortened command with a custom amount of columns:   
Gps| fw -property name -column 4

Another way to view information is in a gridview, this will display in a separate window and can be handy tool to display info outside the PowerShell.

Get-Process | Out-GridView

Storing data:

Here are some handy examples of storing data to other formats:

**.txt**

A simple .txt file:

Get-Volume >>c:\fso\volumeinfo.txt

To overwrite a .txt

Get-Volume >c:\fso\volumeinfo.txt

Use some the following options to make the file more readable:

Get-Service | Format-Table -Property \* -Force -Auto | Out-File -FilePath c:\fso\WideServices.txt -Encoding UTF8 -Width 500

**.csv**

Get-Process | Export-Csv -Path c:\fso\process.csv -NoTypeInformation

Storing and reading from an .csv

Get-Process | Export-Csv -Path c:\fso\processInfo.csv

Import-Csv -Path C:\fso\processInfo.csv | sort vm | select -First 2 | ft name, vm

**.xml**

Make the variable $csv

$csv = Import-Csv -Path C:\fso\processInfo.csv

Retrieving the name:

$csv[0].name

Armsvc

The VM:

$csv[0].vm

46989312

Create an .xml

Get-Process | Export-Clixml -Path c:\fso\processXML.xml

Import an .xml to variable:

$xml = Import-Clixml -Path C:\fso\processXML.xml

# Remote Control

* 1. **What is remote control?**

Remote control is the term to control another computer by using PowerShell.  
 When running Powershell as administrator, you can transmit any command to

a remote computer for local execution. By default this function is disabled but

by enable PSRemoting command u can make it work immediately.  
The original goal of remote session was to control a machine which is not physically located in the office.

* 1. **PSRemoting**The following cmdlet gives you access to another computer (make sure u are administrator to execute this).

PS C:\> Enable-PSRemoting

* 1. **PSSession**

When the PSremoting is enabled, we now have to make sure it works by

entering another computer.  
 PS C:\> Enter-PSSession –ComputerName driscoll-host

–Credential ard\adriscoll -ConfigurationName

Microsoft.Powershell

The -ComputerName is a necessary parameter, while the -Credential

and -ConfigurationName are both optional.

Whenever a user connects to a remote machine, the WinRM service creates a

new instance.  
 Looking up the configuration of the existing session:  
 PS C:\ > Get-PSSessionConfiguration

Modifying existing session configurations:  
 PS C:\ > Set-PSSessionConfiguration Microsoft.Powershell

–ShowSecurityDescriptorUI

When executing this command, there will show up a confirmation prompt that

notify us that the WinRM(Windows Remote Management) will have to be restarted to make the changes happen. It’s also possible to set some permissions for particular groups or users. The permissions are read, write and execute.

Create a new configuration and ask for users to be able to access the session:

PS C:\> Register-PSSessionConfiguration –Name TfsDevelopers -RunAsCredential mdnvdi\tfsbuilder -ShowSecurityDescriptorUI

Create a new PSSession and name it:  
PS C:\> New-PSSession –ComputerName driscoll-desk –Name

CheckMemorySession

We made a script on the name CheckMemoryUsage (scripts boek Daan), we will now use the Invoke-Command to link the variable to the script.

* 1. **Invoke-Command**Invoke-Command runs commands on a local or remote computer and returns also the output from the commands including errors.

We set a variable to the PSSession we made above.  
 PS C:\> $s = Get-PSSession –ComputerName

driscoll-desk–Name CheckMemorySession

Use Invoke-Command so the variable $s runs the script.  
 PS C:\> Invoke-Command –Session $s –FilePath

$Home\CheckMemoryUsage.ps1

Close the session:

PS C:\> Disconnect-PSSession $s

Restart the session:  
 Enter-PSSession –Id 6

Remove session:  
 PS C:\> Remove-PSSession –ComputerName driscoll-desk

–Name CheckMemorySession

Use of idle timeout as alternative to remove the session:  
 PS C:\> $o = New-PSSessionOption –IdleTimeout 300000

PS C:\> New-PSSession –ComputerName badservice-host

–Name CheckMemorySession –SessionOption $o

OR

PS C:\> $PSSessionOption.IdleTimeout = 300000

PS C:\> New-PSSession –ComputerName driscoll-desk –Name

CheckMemorySession

By running these 2 commands, we set the IdleTimeout to 300,000 milliseconds

(5 minutes). This means that the session will end or automatically disconnect

after being idle for 5 minutes.

* 1. **Troubleshooting**

Test as who you are logged on at the remote machine:  
 PS C:\> whoami

# Scripts, the knowhow

* 1. **Why use scripts?**

Automatisation. Yes you can do a lot with one-liners and yes you might do fine without scripts, but you’ll be a lot slower than someone who uses scripts. Scripts offer a way to make the like of a system administrator easy. You don’t even have to be a good scripter, because there are a tons of scripts already out there that could make your job easier.

* 1. **Execution policy**

It’s important to know which execution policy you have running in your PowerShell. Your execution policy decides whether a script will be executed or not.

To know which execution policy is currently applying to you:

Get-ExecutionPolicy

Different options for Set-ExecutionPolicy:

* Restricted

No configuration files or scripts

* AllSigned

Requires that a trusted publisher signs all scripts and configuration files

* RemoteSigned

Requires a trusted publisher sign all scripts and configuration files

* Unrestricted

Requires all configuration files and runs all scripts, if the script comes from the internet you'll be prompted

* Bypass

Nothing is blocked and no prompts!

* Undefined

Removes the current assigned execution policy

In addition to these 6 policy levels there are three different scopes:

* Process

This makes the policy only affect processes

* CurrentUser

The execution policy affects only the current user.

* LocalMachine

The execution policy affects all users of the computer.

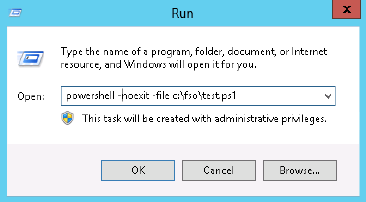
To change the ExecutionPolicy:

Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy unrestricted

* 1. **How do I use PowerShell scripts?**

To run a Windows PowerShell script from inside the Windows PowerShell console, type the full path to the script. Include the name of the script. Ensure you include the ps1 extension.

If you need to run a script from outside Windows PowerShell, you need to type the full path to the script, but you must feed it as an argument to the PowerShell.exe program. In addition, you probably want to specify the -noexit argument so that you can read the output from the script.



To create a script you can work with PowerShell ISE and press new, then write the script and save the file as a .ps1 file. For example a script that stops all instances of a process and reports:

StopProcessSilentlyContinuePassThru.ps1

$process = "notepad"

Get-Process -name $Process -erroraction silentlycontinue |

Stop-Process -passthru |

ForEach-Object { $\_.name + ' with process ID: ' + $\_.ID + ' was stopped.'}

Note: you can change the content of the variable process which makes for an adaptive script which you can easily edit and reuse.

* 1. **Using variables**

An important part of scripting is using variables this makes it so you don’t have to edit your script all the time. By default, when working with Windows PowerShell, you don’t need to declare variables before use. When you’re using a variable to hold data, it is declared. All variable names must be preceded with a dollar sign ($) when they are referenced. There are a number of special variables in Windows PowerShell.  
These variables are created automatically and have a special meaning.

  
*A list of predefined variables*

To use variables in a script you simply create them at the start of the script. As you change the variables of the script you will change the output. For example:

# ReadUserInfoFromReg.ps1

$strUserPath = "\Software\Microsoft\Windows\CurrentVersion\" + "Explorer"

$strUserName = "Logon User Name"

$strPath = "\Volatile Environment"

$strName = "LOGONSERVER","HOMEPATH", "APPDATA","HOMEDRIVE"

Set-Location HKCU:\

Get-ItemProperty -path $strUserPath -name $strUserName |

Format-List $strUserName

foreach ($i in $strName)

{Get-ItemProperty -path $strPath -name $i | Format-List $i}

# Windows Server

* 1. **Deploy a Forest and domain**

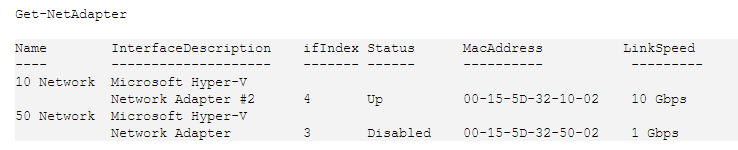
Before you begin, make sure you have identified all the elements you’ll need to configure as you set up the server you’ll use to create your new forest and domain, and what the values for those are. The exact list you’ll need will vary depending on the results of the preliminary planning you’ve done, and your network configuration, but it will likely include at least the following:

* Server IP address
* Server name
* Domain Name System (DNS) namespace for the root domain of the new forest
* Domain name for the root domain of the new forest
* DNS server type (Active Directory–integrated, or stand-alone)
* Webserver (IIS) - Necessary for remote control

A comment here about the server IP address: your domain controllers should ideally all use static IP addresses, but definitely your first domain controller should be at a fixed IP address.

##### Get the adapter alias and index

Before you can configure new settings for a network adapter, you need to know either the adapter’s *interface alias* (name) or *interface index*. The interface alias corresponds to the name shown in the Network Connections dialog box (ncpa.cpl). To determine the interface alias and interface index, use the Get-NetAdapter cmdlet.



The default output from Get-NetAdapter uses the Name column for the InterfaceAlias property and the ifIndex column for the InterfaceIndex property.

Return the networkbindings that match the display:  
PS C:\ > Get-NetAdapterBinding -DisplayName \*TCP/IPv4\*

Get the properties: Get-NetAdapterAdvancedProperty

Set the properties: Set-NetAdapterAdvancedProperty  
Make new properties: New-NetAdapterAdvancedProperty

**Examples:**PS C:\> Get-NetAdapterAdvancedProperty –DisplayName "Receive Buffers"|

Select-Object -ExpandProperty RegistryValue

1. Set a fixed IP address

To set a fixed IP address for this first domain controller in the forest, you need to first disable Dynamic Host Configuration Protocol (DHCP) and then set the IPv4 and IPv6 addresses. For the lab network used in this book, I have chosen 192.168.10.0/24 as the IPv4 subnet, and 2001:db8:0:10::/64 as the IPv6 subnet.

To disable DHCP on the 10 Network adapter, use the following command.

Set-NetIPInterface -InterfaceAlias "10 Network" -DHCP Disabled -PassThru

Next, set the static IPv4 address to 192.168.10.2 by using the following command.

New-NetIPAddress `

-AddressFamily IPv4 `

-InterfaceAlias "10 Network" `

-IPAddress 192.168.10.2 `

-PrefixLength 24 `

-DefaultGateway 192.168.10.1

Now set the IPv6 address to 2001:db8:0:10::2 by using the following command.

New-NetIPAddress `

-AddressFamily IPv6 `

-InterfaceAlias "10 Network" `

-IPAddress 2001:db8:0:10::2 `

-PrefixLength 64 `

-DefaultGateway 2001:db8:0:10::1

The New-NetIPAddress cmdlet automatically selects the IPv4 or IPv6 address family based on the settings in the command, so you can omit the -AddressFamily parameter from the preceding commands if you want.

1. Set the DNS server addresses

The last part of setting a fixed IP address is to set the DNS server addresses. Because your first domain controller in the new forest should also be your DNS server, that’s pretty easy to do by using the Set-DnsClientServerAddress cmdlet.

Set-DnsClientServerAddress `-InterfaceAlias "10 Network" `

-ServerAddresses 192.168.10.2,2001:db8:0:10::2

1. Check settings

Get-NetIPAddress -InterfaceAlias "10 Network"

1. Set the server name

Before you actually deploy your new forest, you should set the name of your domain controller to match your naming convention. Changing the name of a computer causes a reboot, which is why you should delay that change until after all the IP address setting is done. To change the name of the new server to trey-dc-02, use the Rename-Computer cmdlet by using the following syntax.

Rename-Computer -NewName trey-dc-02 -Restart -Force -PassThru

This changes the name of the server and automatically restarts it. The -Force parameter suppresses the confirmation prompt, and the -PassThru parameter returns the results of the command. After the server restarts, you’re ready to actually deploy your forest.

1. Install ADDS

Before you can promote the server to be a domain controller, you need to install the Active Directory Domain Services role on the server. Installing a role or feature uses the Install-WindowsFeature cmdlet.

Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools

The actual command to install the new forest and domain is:  
 Install-ADDSForest `

-DomainName 'TreyResearch.net' `

-DomainNetBiosName 'TREYRESEARCH' `

-DomainMode 6 `

-ForestMode 6 `

-NoDnsOnNetwork `

-SkipPreChecks `

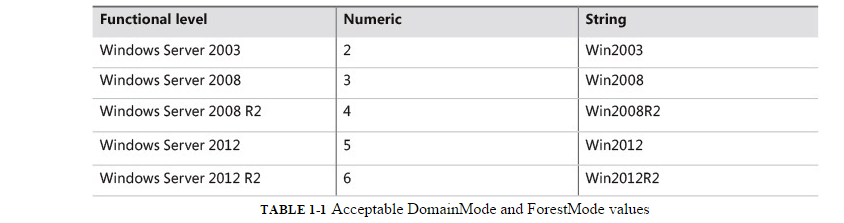
-Force

The other thing added here is a -Force parameter to suppress any confirmation prompts. You’ll still be prompted for the value of the Directory Services Restore Mode (DSRM) password. You can avoid even that by using the -SafeModeAdministratorPassword parameter with a SecureString value equivalent to your password.

If you’re automating a lot of forest (or domain) creations, such as in a lab environment, use this syntax to set the DSRM password to a value of P@ssw0rd!.

$pwdSS = ConvertTo-SecureString -String 'P@ssw0rd!' -AsPlainText -Force

The acceptable values for ForestMode and DomainMode are shown in Table 1-1.



When you create the new forest, the server is rebooted, and the only account active on the server is the TREYRESEARCH\Administrator account, which has the same password as the safe mode password you used with Install-ADDSForest.

To find out what Forest Mode, Domain Mode, and Schema Version you’ve just created, use the following script:

[CmdletBinding()]

Param(

[Parameter(Mandatory=$False,Position=0)]

[string]

$DomainController

)

if ($DomainController) {

$AD = Get-ADRootDSE -Server $DomainController

Get-ADObject $AD.SchemaNamingContext -Server $DomainController `

-Property ObjectVersion

} else {

$AD = Get-ADRootDSE

Get-ADObject $AD.SchemaNamingContext -Property ObjectVersion

}

$Forest = $AD.ForestFunctionality

$Domain = $AD.DomainFunctionality

# Use a Here-String to print out the result.

$VersionCodes = @"

Forest: $Forest

Domain: $Domain

Where the Schema version is:

72 = Windows Server Technical Preview Build 9841

69 = Windows Server 2012 R2

56 = Windows Server 2012

47 = Windows Server 2008 R2

44 = Windows Server 2008

31 = Windows Server 2003 R2

30 = Windows Server 2003

13 = Windows 2000

"@

$VersionCodes

Install-ADDSForest has some additional options that might be useful in your environment and that allow you to tweak the initial configuration. Table 1-2 shows a fuller list of the options for Install-ADDSForest.



7) Install Powershell Web Access

PS C:\> Install-PSWAWebApplication –UseTestCertificate

* 1. **Manage DNS and DHCP**

In this chapter, you’ll learn how to create and manage DNS zones, including primary, secondary, and stub zones, and how to create and manage DNS records. You’ll also learn basic DHCP setup and configuration.

1. Windows Server DNS basics

A primary DNS zone provides authoritative name resolution for the zone. In traditional DNS, there is a single primary zone and as many secondary zones as needed to support the name resolution traffic for the zone. Windows Server supports traditional primary and secondary DNS zones and stub zones but uses an Active Directory–integrated, primary DNS zone by default. Because Active Directory integration is used, the zone can be stored on all AD DS domain controllers, providing a high level of fault tolerance and distributing the name resolution traffic across all domain controllers.

Secondary DNS zones are read-only zones that help distribute the network traffic and provide faster name resolution. They contain full copies of all DNS records for the zone. They are primarily useful with traditional primary DNS zones.

A stub DNS zone doesn’t contain full DNS information for the zone, only the necessary information about which servers are authoritative for the zone. Stub zones are useful when you don’t want to expose all the details of a particular zone but still need to provide name resolution. The stub zone has records only for the primary DNS servers for the zone. When a DNS request for the zone is received, the stub zone queries the name servers it has in the stub zone to answer the DNS request.

In the Chapter 1, “Deploy your first forest and domain,” when you deployed your initial AD DS domain and forest, an Active Directory–integrated primary DNS zone was created and configured for the new domain automatically. This was a forward lookup zone, that is a zone that allows name lookups and returns IP addresses. The forest creation process does not, however, create reverse lookup zones that allow IP address lookups that return the machine name.

You can create additional DNS zones and configure them either as Active Directory–integrated or to use stand-alone zone files. Secondary DNS zones always use stand-alone zone files. When you configure a zone by using zone files, the zone is stored in those zone files. The default location for DNS zone files is %windir%\system32\dns, with a file name of zonename.dns, where zonename is the name of the zone, such as TreyResearch.net.

1. Create and manage primary zones

You can create new zones for both Active Directory–integrated and file-based zones. Use the Add-DnsServerPrimaryZone cmdlet to create new zones. To create an Active Directory–integrated **primary forward lookup zone** for TailspinToys.com, use the following command.

Add-DnsServerPrimaryZone -Name 'tailspintoys.com' `

-ComputerName 'trey-dc-02.treyresearch.net' `

-ReplicationScope 'Domain' `

-DynamicUpdate 'Secure' `

-PassThru

Creating a **reverse lookup zone** is similar to creating a forward lookup zone. The difference is that instead of using the Name parameter, you use the NetworkID parameter. Thus, to create an Active Directory–integrated reverse lookup zone that is replicated across the entire forest and that accepts both secure and nonsecure dynamic updates, use the following command.  
  
Add-DnsServerPrimaryZone -NetworkID 192.168.10.0/24 `

-ReplicationScope 'Forest' `

-DynamicUpdate 'NonsecureAndSecure' `

-PassThru

The NetworkID parameter deserves a bit more detail. The parameter expects an ID in the A.B.C.D/prefix format for IPv4 or the 1111:2222:3333:4444::/prefix format for IPv6. Only A, B, C, or D class zones can be created for IPv4; partial zones are not supported. For IPv6, zone prefixes of /16 to /128 in 4-bit increments are supported.

To create an IPv6 reverse lookup zone for TreyResearch.net’s IPv6 range, use the following command.

Add-DnsServerPrimaryZone -NetworkID 2001:db8:0:10::/64 `

-ReplicationScope 'Forest' `

-DynamicUpdate 'Secure' `

-PassThru

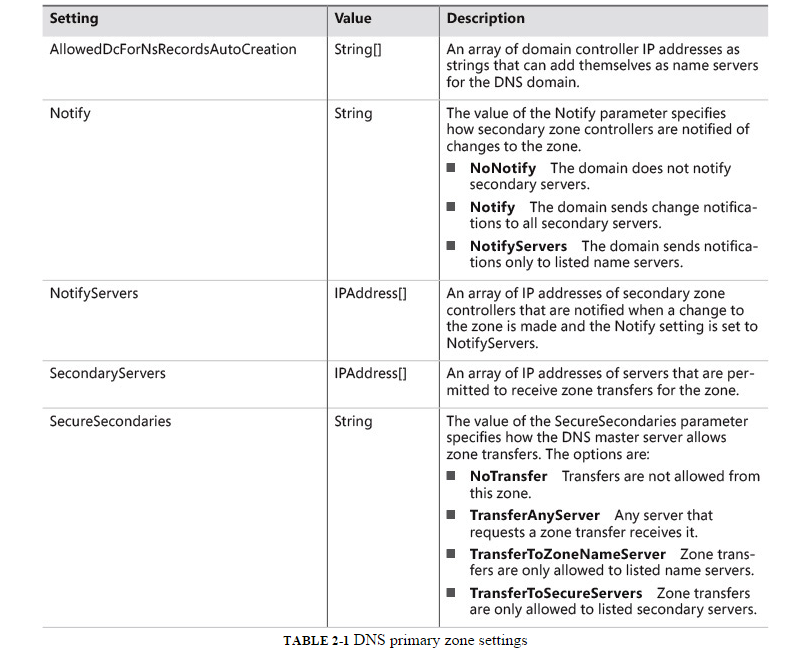
Creating **file-based zones** uses the -ZoneFile parameter. For example, see the following code.

Add-DnsServerPrimaryZone -Name 'tailspintoys.com' `

-ZoneFile 'tailspintoys.com.dns' `

-DynamicUpdate 'None'

When you create a new primary DNS zone, some settings are simply set to their default values, and there aren’t specific parameters to Add-DnsServerPrimaryZone so that those values can be configured at creation. However, you can use the Set-DnsServerPrimaryZone cmdlet to configure those settings. Settings that can be configured after creation are shown in Table 2-1.



1. Create and manage secondary dns zones

Secondary DNS zones are primarily used for providing distributed DNS resolution when you are using traditional file-based DNS zones. Secondary DNS zones are used for both forward lookup and reverse lookup zones. The DnsServerSecondaryZone set of cmdlets is used to deploy and manage secondary DNS zones.

A secondary DNS zone is a read-only zone and depends on transferring the data for the zone from another DNS server. That other server must be configured to allow zone transfers.

Use the Add-DnsServerSecondaryZone cmdlet to **create a new secondary DNS** zone. You can use this command to create either forward lookup or reverse lookup zones. To create a secondary reverse lookup zone for the IPv6 zone created and exported earlier, copy the exported DNS file to %windir%\system32\dns on the server you want to add a secondary DNS zone to, and then use the following command.

Add-DnsServerSecondaryZone –Name 0.1.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa `

-ZoneFile "0.1.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa.dns" `

-LoadExisting `

-MasterServers 192.168.10.2,2001:db8:0:10::2

-PassThru

Secondary DNS zones have few settings because they are read-only zones dependent on their master servers. When you create a new secondary DNS zone, all of the settings for the zone can be set as part of the Add-DnsServerSecondaryZone cmdlet.   
However, if you later need to change those values (to change the master servers, for example), you can use the Set-DnsServerSecondaryZone cmdlet to **modify the zone settings.** To modify the same IPv6 reverse lookup secondary zone to use a different array of master servers, use the command.

Set-DnsServerSecondaryZone -Name 0.1.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa `

-MasterServers 192.168.10.3,2001:db8:0:10::3 `

-PassThru

A secondary zone can only transfer the zone from a primary zone that allows the transfer, preventing unauthorized zone transfers. The setting of zone transfer permissions is done at the primary zone, not at the secondary zone. To **set the zone transfer permissions**, use theSet-DnsServerPrimaryZone cmdlet. The SecureSecondaries parameter controls the servers to which transfers are allowed. The SecureSecondaries parameter accepts the following values:

* **NoTransfer** No transfers are allowed for this zone from this server.
* **TransferAnyServer** Any server can request a zone transfer, including servers that you know nothing about and don’t manage or control.
* **TransferToZoneNameServer** Only servers in the NS records for this zone are allowed to request transfers.
* **TransferToSecureServers** Only servers specified with the -SecondaryServers parameter are allowed to request a zone transfer.

To set the TreyResearch.net zone to only allow zone transfers to name servers, use the following command.

Set-DnsServerPrimaryZone -Name 'treyresearch.net' `

-SecureSecondaries TransferToZoneNameServer `

-PassThru

1. Manage stub zones

Stub DNS zones contain the necessary records to locate a zone’s name servers *only*. They do not maintain the records of other devices or computers in the zone. They’re a useful way to keep track of which servers are currently authoritative for a child zone, without maintaining full records for that child zone. Stub zones can be file-based or Active Directory–integrated and can be used for both forward lookup zones and reverse lookup zones.

Secondary zones are a significant security concern, because they expose all of the records of the zone, potentially providing an attacker with important information that would facilitate further attack. With stub zones, the only exposure is the names and IP addresses of the designated name servers. Further, stub DNS zones don’t require that the primary zone allow zone transfers. The DnsServerStubZone pair of cmdlets is used to deploy and manage stub zones. **To add a new stub zone** for TailspinToys.com, whose master server is 192.168.10.4, **and replicate that stub zone across the domain**, use the following.

Add-DnsServerStubZone -Name TailspinToys.com `

-MasterServers 192.168.10.4 `

-ReplicationScope Domain `

-PassThru

To change the properties of the DNS stub zone for TailspinToys.com to use a local array of master servers (stored in the Windows registry), use the following.

Set-DnsServerStubZone -Name TailspinToys.com `

-LocalMasters 192.168.10.201,192.168.10.202 `

-PassThru

1. Manage DNS records

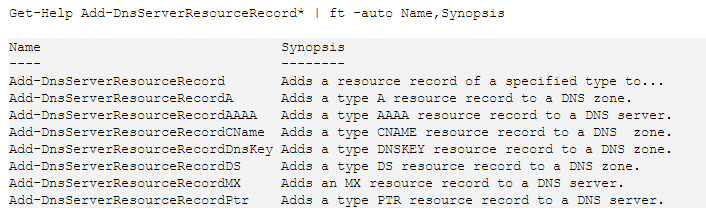
DNS servers do more than simply translate a computer name into an IP address, though that’s certainly their first and primary duty. They also provide the information that other services and servers need to know which server hosts a particular service. So, for example, Internet mail servers need to know which server in TreyResearch.net is the mail server for all of TreyResearch.net’s email. And other clients and servers on the TreyResearch network need to know which servers are official name servers for the zone.

Each of these services is designated by a specific type of DNS *resource record*. The basic A or AAAA resource record translates a DNS name into an IPv4 or IPv6 address, respectively. An MX resource record specifies a mail server for the domain, and NS resource records are used to specify which servers are name servers for the domain.

Windows DNS supports a wide variety of DNS records, but the core resource records are the following:

* **A** An IPv4 host address record. The A record is a forward lookup record that translates a host name into an IPv4 address.
* **AAAA** An IPv6 host address record. The AAAA record is a forward lookup record that translates a host name into an IPv6 address.
* **CNAME** A canonical name record. The CNAME record facilitates the use of more than one resource record to refer to a single host.
* **DS** A Delegated Signer record. The DS record is used with Domain Name System Security Extensions (DNSSEC) to designate the subzone signing key.
* **DNSKey** The public key record for a DNSSEC signed zone.
* **MX** A Mail Exchanger record. The MX record identifies the email server for a domain. There can be multiple MX records for a domain; they are used in order of precedence.
* **NS** A Name Server record. The NS record identifies a name server for the domain. There can be multiple NS records in a domain.
* **PTR** A pointer record. The PTR record is a reverse lookup record that translates an IP address into a host name. PTR records can be IPv4 or IPv6 addresses.
* **TXT** A text record. The TXT record is used to assign unformatted text to a host in DNS. A common use of TXT records is for Sender Policy Framework (SPF) records used to identify legitimate email senders.
* **SOA** A Start of Authority record. The SOA record is a version number record identifying the version number of the DNS zone.
* **SRV** A Service record. The SRV record identifies the host name and port number of servers for the specified service.

All of the resource records listed here can be created or changed by using the Windows PowerShell DnsServerResourceRecord cmdlets with the exception of the SOA record. SOA records are automatically updated whenever a change is made to the DNS zone. In addition to the general Add-DnsServerResourceRecord cmdlet, which can be used to make all supported resource records, there are specific Add- cmdlets for the A, AAAA, CNAME, DS, DNSKey, MX, and PTR resource records.



The process for **creating any DNS resource record** is essentially the same, though each record type has settings appropriate to the type. So, for example, to create an A record for the server trey-wds-11, use the following command

Add-DnsServerResourceRecord -ZoneName "TreyResearch.net" `

-A `

-Name trey-wds-11 `

-IPv4Address 192.168.10.11 `

-CreatePtr `

-PassThru

This uses the general Add-DnsServerResourceRecord cmdlet, therefore I need to specify the record type (-A). And because I’m also running a reverse-lookup zone, I added the CreatePtr parameter to automatically create the PTR resource record for this server. The results are displayed back to the console because I used the PassThru parameter.

Using the specific Add-DnsServerResourceRecordA cmdlet to create the same DNS A record would use the following command.

Add-DnsServerResourceRecordA -ZoneName "TreyResearch.net" `

-Name trey-wds-11 `

-IPv4Address 192.168.10.11 `

-CreatePtr `

-PassThru

The CNAME record, sometimes called an alias, is a canonical record you can use to add an additional host name to a server. The CNAME record points to an existing host name record. When the DNS server receives a name query for the name in the CNAME record, it looks up the A record for the host name that the CNAME points to and returns the IPv4 address for that host name.

**To create a CNAME record** for server WDS that points to the A record for trey-wds-11, use the following command.

Add-DnsServerResourceRecord -ZoneName "TreyResearch.net" `

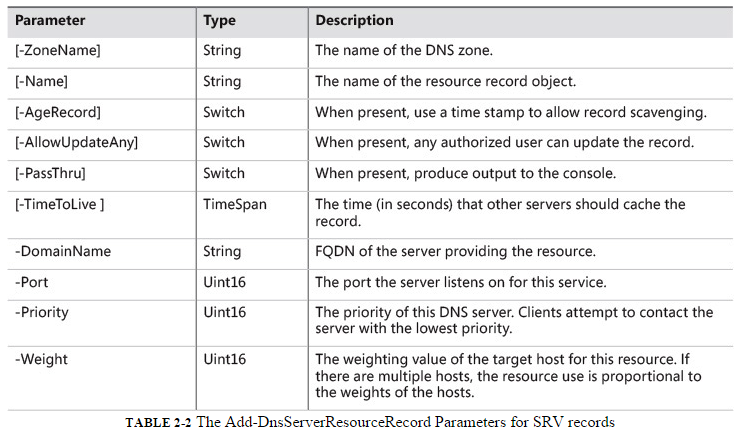
-CName `

-Name wds `

-HostNameAlias trey-wds-11.treyresearch.net

-PassThru

The basic syntax for creating any DNS resource record by using the Add-DnsServerResourceRecord cmdlet is the same. What varies is the specific parameters for the resource type. Use the Get-Help pages (or the Get-Syntax.ps1 script I provided earlier) to get the specific parameters for the record type you’re creating. For example, the SRV record type supports the parameters shown in Table 2-2.



1. Manage DHCP

DHCP is the method used to dynamically assign IP addresses to computers and devices on a Windows network. DHCP can be deployed on servers running Windows Server, or on other devices or servers on the network, but for most Windows AD DS domains, a server running Windows Server and the DHCP service is preferred. Plus, that lets us use Windows PowerShell to deploy and configure the DHCP.

Depending on the size and complexity of your Windows network, you might have a single DHCP server or many, and they can be stand-alone or clustered. But this isn’t a book about DHCP, so I’m going to cover just the basics of setting up a single DHCP server on a lab-sized 192.168.10.0/24 network, and on the corresponding 2001:db8::10:0/64 network.

To deploy DHCP, you need to first install the role. For this, use the server trey-dns-03 at 192.168.10.3 as the target.

Install-WindowsFeature -ComputerName trey-dns-03 `

-Name DHCP `

-IncludeAllSubFeature `

-IncludeManagementTools

No surprises there, and actually the IncludeAllSubFeature parameter isn’t really needed, because DHCP doesn’t have any subfeatures. But it’s a good habit to get into, and it won’t complain or throw an error if there aren’t any subfeatures. Notice here that I’m running this as a remote command. I’m actually running the command from an elevated Windows PowerShell window on trey-dc-02, but I want the DHCP to be on a different server. As you can tell from Figure 2-7, the installation was a success. But that’s just barely the start. Now you need to actually do the initial setup of the server and activate it in AD DS.

Next, you need to create two local groups on the DHCP server, DHCP Administrators and DHCP Users. You can do this by using lusrmgr.msc or by using the following commands.

#The WinNT in the following IS CASE SENSITIVE

$connection = [ADSI]"WinNT://trey-dns-03"

$lGroup = $connection.Create("Group","DHCP Administrators")

$lGroup.SetInfo()

$lGroup = $connection.Create("Group","DHCP Users")

$lGroup.SetInfo()

The final step before you start configuring the DHCP server is to activate it in AD DS as an authorized server. To do this, use the Add-DhcpServerInDC cmdlet. Because you’re still running remotely, you need to specify which server to add. You could use either the DNS name or the IP address. The command is.

Add-DhcpServerInDC -DnsName 'trey-dns-03' -PassThru

**Note:**  
If you want to use cmdlets in the DHCP module on a computer that is not running DHCP, you need to install the RSAT tools first, by using the following.

Install-WindowsFeature -Name RSAT-DHCP

1. Configure DHCP: IPv4

For IPv4, let’s start by creating an initial scope. For my purposes, a scope of 200 addresses makes sense, leaving me 50 or so to use for a second scope on a separate server if I want to create some simple redundancy.  
The command to create an IPv4 scope is Add-DhcpServerv4Scope, and again, you can run it from a remote computer. You need to assign a start for the scope and an end point, set the subnet mask, and set the scope to active. The command is as follows.

Add-DhcpServerv4Scope -Name "Trey-Default" `

-ComputerName "trey-dns-03" `

-Description "Default IPv4 Scope for Lab" `

-StartRange "192.168.10.1" `

-EndRange "192.168.10.200" `

-SubNetMask "255.255.255.0" `

-State Active `

-Type DHCP `

-PassThru

Next, let’s set an exclusion range so that we can have some fixed IP address servers. For this, use the Add-DhcpServerv4ExclusionRange cmdlet. This uses the ScopeID to identify which scope is being configured. The command to exclude addresses from 192.168.10.1 through 192.168.10.20 is as follows.

Add-DhcpServerv4ExclusionRange -ScopeID "192.168.10.0" `

-ComputerName "trey-dns-03" `

-StartRange "192.168.10.1" `

-EndRange "192.168.10.20" `

-PassThru

Next, set the options on your scope, by using the Set-DhcpServerv4OptionValue cmdlet. Then you should be good to go for IPv4.

Set-DhcpServerv4OptionValue -ScopeID 192.168.10.0 `

-ComputerName "trey-dns-03" `

-DnsDomain "TreyResearch.net" `

-DnsServer "192.168.10.2" `

-Router "192.168.10.1" `

-PassThru

1. Configure DHCP: IPv6

To configure the IPv6 scope, use the Add-DhcpServerv6Scope cmdlet. You need to assign a start for the scope and an end point, set the Prefix (subnet mask), and set the scope to active. The command is as follows.

Add-DhcpServerv6Scope -Name "Trey-IPv6-Default" `

-ComputerName "trey-dns-03" `

-Description "Default IPv6 Scope for Lab" `

-Prefix 2001:db8:0:10:: `

-State Active `

-PassThru

Next, set an exclusion range for the servers. Here, we’re carving out 32 addresses.

Add-DhcpServerv6ExclusionRange –ComputerName trey-dns-03 `

-Prefix 2001:db8:0:10:: `

-StartRange 2001:db8:0:10::1 `

-EndRange 2001:db8:0:10::20 `

Next, set the options on your scope by using the Set-DhcpServerv6OptionValue cmdlet to set the Scope options for IPv6.

Set-DhcpServerv6OptionValue -Prefix 2001:db8:0:10:: `

-ComputerName "trey-dns-03" `

-DnsServer 2001:db8:0:10::2 `

-DomainSearchList "TreyResearch.net" `

-PassThru

And we’re done with a basic DHCP setup. There’s lots more you can configure for the DHCP server role. Use the Set-DhcpServer set of cmdlets to configure the additional options.

* 1. **Create and manage usergroups**

Now that we have a forest and domain, and we’ve got the basics of networking and name resolution sorted, the next step is to add some users to our domain. We’ll start with adding a simple user, interactively, and then create a bunch of users by using a script and a comma-separated values (CSV) file. We’ll create a new group and then add a group of users into that group, using a filter to ensure that we add the correct set of users. Then we’ll create a new organizational unit (OU) and move users and computers into the OU. Pretty basic stuff, really, but essential for any domain administrator.

#### **Create a single user**

The first thing you’ll want to do for your new domain is create an administrative user that isn’t “Administrator.” That first Administrator account is sometimes referred to as the 500 account because the last three digits of its security identifier (SID) are 500, as we can tell from a quick Get-ADUser.

Get-ADUser -Identity Administrator

The 500 account is a bit too well known to use for everyday administration and should be given a really long and onerous password that is locked away somewhere very secure and then left alone except in dire emergencies. So let’s give ourselves a working administrative account, and then we’ll change the password on the 500 account and retire it from everyday use.

To add a new user, use the New-ADUser cmdlet. There are three basic ways to use New-ADUser:

**1.** Create a user by specifying all details on the command line.

**2.** Create a user from a template object—either one you create or an existing user.

**3.** Use a CSV file to create multiple users from a list of users and properties.

We’re going to use option #1 to create our first administrative user. We need to specify the settings for the new user at the command line. Then we need to add the user to the appropriate Active Directory Domain Services (AD DS) security groups.

First, to create the user, “Charlie,” use the following commands.

$SecurePW = Read-Host -Prompt "Enter a password" -asSecureString

New-ADUser -Name "Charlie Russel" `

-AccountPassword $SecurePW `

-SamAccountName 'Charlie' `

-DisplayName 'Charlie Russel' `

-EmailAddress 'Charlie@TreyResearch.net' `

-Enabled $True `

-GivenName 'Charlie' `

-PassThru `

-PasswordNeverExpires $True `

-Surname 'Russel' `

-UserPrincipalName 'Charlie'

This creates our first user but doesn’t make the user a member of any domain security groups except Domain Users, the default. To add the user to security groups, we need to use the Add-ADGroupMember cmdlet. And because the goal is to give Charlie the same set of security groups as the Administrator account, we’ll use Windows PowerShell to get the list of security groups that the Administrator is a member of, and then loop through the list and add Charlie to each of the groups.

$SuperUserGroups = @()

$SuperUserGroups = (Get-ADUser -Identity "Administrator" -Properties \* ).MemberOf

ForEach ($Group in $SuperUserGroups ) {

Add-ADGroupMember -Identity $Group -Members "Charlie"

}

(Get-ADUser -Identity Charlie -Properties \*).MemberOf

As we can tell from the Get-ADUser command in the previous code, the account Charlie is now a member of five security groups: Group Policy Creator Owners, Domain Admins, Enterprise Admins, Schema Admins, and Administrators. These are the same security groups to which the Administrator account belongs. We’ll want to come back to AD DS groups later, but let’s focus on users first.

#### **Create users in a batch**

There are multiple ways to add users in a batch, but probably the simplest is to use a CSV file. You can easily create the CSV file in Microsoft Excel or any plain text editor, and then use Windows PowerShell to read the values in the CSV file and add the users. In my lab, all my animals have their own domain accounts, so I’ll use them to show how to quickly and easily create new users. All are initially created as Domain Users, with a default password, and then one account gets elevated and prompts for a password. The list of users and their basic properties are in the following code.

*TreyUsers.csv:*

**Name,GivenName,Surname,DisplayName,SAMAccountName,Description**

**David Guy,David,Guy,Dave R. Guy,Dave,Customer Appreciation Manager**

**Alfredo Fettucine,Alfredo,Fettuccine,Alfie NoNose,Alfie,Shop Foreman**

**Stanley Behr,Stanley,Behr,Stanley T. Behr, Stanley,WebMaster**

**Priscilla Catz,Priscilla,Catz,Dame Priscilla,Priscilla,Shop Steward**

**Harold Catz,Harold,Catz,Harold S. Catz,Harold,Engineering Manager**

**William Wallace,William,Wallace,Sir William Wallace,Wally,Marketing Manager**

**Trey Barksdale,Trey,Barksdale,Lord Barksalot,Trey,Sales Manager**

**Charlie Russel,Charlie,Russel,Charlie Russel,Charlie,SuperUser Account**

As you can tell, I’ve only used the most basic information for each new user. To read the CSV file, use the Import-CSV cmdlet, and then loop through each user from the CSV file and create the user with New-ADUser by using a basic ForEach loop.

*Create-TreyUsers.ps1:*

**<#**

**.Example**

**Creates AD Accounts for the users in the default "TreyUsers.csv" source file**

**.Example**

**Create-TreyUsers -Path "C:\temp\NewUsers.txt"**

**Creates AD accounts for the users listed in the file C:\temp\NewUsers.txt"**

**.Parameter Path**

**The path to the input CSV file. The default value is ".\TreyUsers.csv".**

**#>**

**[CmdletBinding()]**

**Param(**

**[Parameter(Mandatory=$False,Position=0)]**

**[string]**

**$Path = ".\TreyUsers.csv"**

**)**

**$TreyUsers = @()**

**If (Test-Path $Path ) {**

**$TreyUsers = Import-CSV $Path**

**} else {**

**Throw "This script requires a CSV file with user names and properties."**

**}**

**ForEach ($user in $TreyUsers ) {**

**New-AdUser -DisplayName $User.DisplayName `**

**-GivenName $user.GivenName `**

**-Name $User.Name `**

**-SurName $User.SurName `**

**-SAMAccountName $User.SAMAccountName `**

**-Enabled $True `**

**-PasswordNeverExpires $true `**

**-UserPrincipalName $user.SAMAccountName `**

**-AccountPassword (ConvertTo-SecureString -AsPlainText -Force -String**

**"P@ssw0rd!" )**

**If ($User.SAMAccountName -eq "Charlie" ) {**

**$cprpwd = Read-Host -Prompt 'Enter Password for account: Charlie' -AsSecureString**

**Set-ADAccountPassword -Identity Charlie -NewPassword $cprpwd -Reset**

**$SuperUserGroups = @()**

**$SuperUserGroups = (Get-ADUser -Identity "Administrator" -Properties \* ).MemberOf**

**ForEach ($Group in $SuperUserGroups ) {**

**Add-ADGroupMember -Identity $Group -Members "Charlie"**

**}**

**Write-Host "The user $user.SAMAccountName has been added to the following AD**

**Groups: "**

**(Get-ADUser -Identity $user.SAMAccountName -Properties \* ).MemberOf**

**}**

**}**

If you want more detail about the individual accounts that you created, modify the New-ADUser command in the script to include the PassThru parameter. With that change, though, you’ll get a lot more detail than you likely want. Instead, try a quick one-line search to find the users.

**(Get-ADUser -Filter {Enabled -eq "True"} -Properties DisplayName).DisplayName**

* 1. **Restart-computer cmd ergens zetten? (p61-64 mijnboek)**
  2. **Powershell workflow with visual studio intigration?p65-96**
  3. **WMI/CMI provider? p97-122**
  4. **Snippet(small scripts to use in ISE) p135**
  5. **Usage of ISE?p136-144**
  6. **Smbshare: p154**

**Scripts boek Daan (ergens in te verwerken?)**Check for memory leak in a process:  
$stopTime = Get-Date

$stopTime.AddMinutes(10)

Start-Job –Name MemoryJob {

while ((Get-Date) –lt $stopTime)

{

Get-Process BadService | Select-Object PrivateMemorySize, WorkingSet,

VirtualMemorySize

Start-Sleep –Seconds 2

}

}

* 1. **sqdfsq**
  2. **qsdfqsdf**
  3. **fdqsqdsf**
  4. **qsdfqdsf**
  5. **qsdfqdsf**
  6. **dqsfqdsf**
  7. **qsdfqsdf**
  8. **qsdffdqs**
  9. **dqsfqdfs**