2

Exchange Management Shell Common Tasks

In this chapter, we will cover:

* Manually configuring remote PowerShell connections
* Using explicit credentials with PowerShell cmdlets
* Transferring files through remote shell connections
* Managing domains or an entire forest using recipient scope
* Exporting reports to text and CSV files
* Sending SMTP e-mails through PowerShell
* Scheduling scripts to run at a later time
* Logging shell sessions to a transcript
* Automating tasks with the scripting agent

Scripting an Exchange Server installation

# Introduction

Microsoft introduced some radical architectural changes in Exchange 2007, including a brand-new set of management tools. PowerShell, along with an additional set of Exchange Server specific cmdlets, finally gave administrators an interface that could be used to manage the entire product from a command line shell. This was an interesting move, and at that time the entire graphical management console was built on top of this technology.

The same architecture still existed with Exchange 2010, and PowerShell was even more tightly integrated with this product. Exchange 2010 used PowerShell v2, which relied heavily on its new remoting infrastructure. This provides seamless administrative capabilities from a single seat with the Exchange Management Tools, whether your servers are on-premises or in the cloud.

Initially when Exchange 2013 was released it was using version 4 of PowerShell, and during the lifecycle it could be updated to version 5 of PowerShell with a lot of new cmdlets, core functionality changes and even more integrations with the cloud services.

Now with Exchange 2016 we have even more cmdlets and even more integrations with cloud related integration and services. During the initial work on this book we had 839 cmdlets with Cumulative Update 4 which was released in December 2016. This can be compared with the previous book at that stage we had 806 cmdlets where it was based on Service Pack 1 and Cumulative Update 7.

It gives us an impression that Microsoft are working heavily on the integrations and that the development of the on-premises product is still ongoing. This demonstrates more features and functionality have been added over time, it will most likely continue like this in the future as well.

In this chapter, we'll cover some of the most common topics, as well as common tasks that will allow you to effectively write scripts with this latest release. We'll also take a look at some general tasks such as scheduling scripts, sending e-mails, generating reports, and more.

## Performing some basic steps

To work with the code samples in this chapter, follow these steps to launch the Exchange Management Shell:

1. Log onto a workstation or server with the Exchange Management Tools installed.
2. You can connect using remote PowerShell if you for some reason don't have Exchange Management Tools installed. Use the following command:

$Session = New-PSSession -ConfigurationName Microsoft.Exchange `

-ConnectionUri http://tlex01/PowerShell/ `

-Authentication Kerberos

Import-PSSession $Session

1. Open the Exchange Management Shell by clicking the windows button and go to Microsoft Exchange Server 2016 | Exchange Management Shell.

Remember to start the Exchange Management Shell using **Run as Admin** to avoid permission problems.   
  
In the chapter, notice that in the examples of cmdlets, I have used the back tick (`) character for breaking up long commands into multiple lines.  
The purpose with this is to make it easier to read. The back ticks are not required and should only be used if needed.

Notice that the Exchange variables, like $exscripts, are not available when using the method above.

# Manually configuring remote PowerShell connections

Just like Exchange 2013, Exchange 2016 is very reliable on remote PowerShell for both on premises and cloud services. When you double-click the Exchange Management Shell shortcut on a server or workstation with the Exchange Management Tools installed, you are connected to an Exchange server using a remote PowerShell session.

PowerShell remoting also allows you to remotely manage your Exchange servers from a workstation or a server even when the Exchange Management Tools are not installed. In this recipe, we'll create a manual remote shell connection to an Exchange server using a standard PowerShell console.

## Getting ready

To complete the steps in this recipe, you'll need to log on to a workstation or a server and launch Windows PowerShell.

## How to do it...

1. First, create a credential object using the Get-Credential cmdlet. When running this command, you'll be prompted with a Windows authentication dialog box. Enter a username and password for an account that has administrative access to your Exchange organization. Make sure you enter your user name in DOMAIN\USERNAME or UPN format:

$credential = Get-Credential

1. Next, create a new session object and store it in a variable. In this example, the Exchange server we are connecting to is specified using the -ConnectionUri parameter. Replace the server FQDN in the following example with one of your own Exchange servers:

$session = New-PSSession -ConfigurationName Microsoft.Exchange `

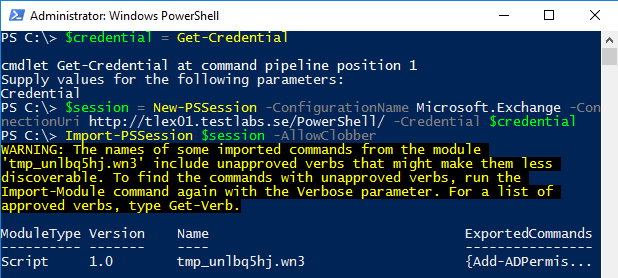
-ConnectionUri http://tlex01.testlabs.se/PowerShell/ `

-Credential $credential

1. Finally, import the session object:

Import-PSSession $session -AllowClobber

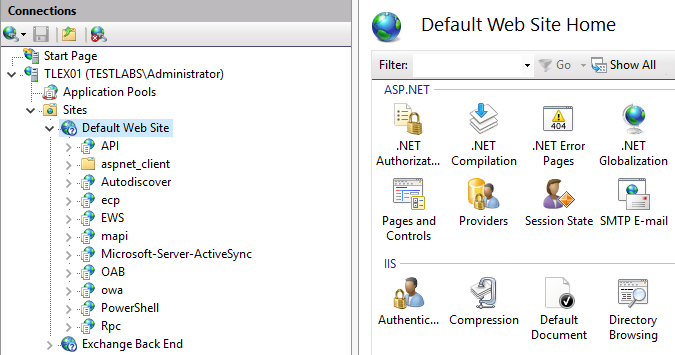
After you execute the preceding command, the Exchange Management Shell cmdlets will be imported into your current Windows PowerShell session, as shown in the following screenshot:



7081EN\_02\_01

## How it works...

Each server runs IIS and supports remote PowerShell sessions via HTTP. Exchange Servers host a PowerShell virtual directory in IIS. This contains several modules that perform authentication checks and determine which cmdlets and parameters are assigned to the user making the connection. This happens both when running the Exchange Management Shell with the tools installed, and when creating a manual remote connection. The IIS virtual directory that is being used for connecting is shown in the following screenshot:



7081EN\_02\_02

The IIS virtual directories can also be retrieved by using PowerShell with the cmdlet   
Get-WebVirtualDirectory and for getting the information about the web applications, use the cmdlet Get-WebApplication.

Remote PowerShell connections to Exchange 2016 servers connect almost the same way as Exchange 2013 did, called implicit remoting that allows us to import remote commands into the local shell session. With this feature, we can use the Exchange PowerShell cmdlets installed on the Exchange server and load the cmdlets into our local PowerShell session without installing any management tools.

However, the detailed behavior for establishing a remote PowerShell session was changed in Exchange 2013 CU11. What happens right now when a user or admin is trying to establish the PowerShell session is that it first tries to connect to the users or admins mailbox (anchor mailbox), if there are any. If the user doesn’t have an existing mailbox, the PowerShell request is being redirected to the organization arbitration mailbox named SystemMailbox{bb558c35-97f1-4cb9-8ff7-d53741dc928c}.

You'll need to allow the execution of scripts in order to create a manual remote shell connection on a machine that does not have the Exchange tools installed. For more details, refer to the Understanding the new Execution Policy recipe in Chapter 1, PowerShell Key Concepts.

You may be curious as to why Exchange uses remote PowerShell even when the tools are installed and when running the shell from the server. There are a couple of reasons for this, but some of the main factors are permissions. The Exchange 2010, 2013 and 2016 permissions model has been completely transformed in these latest versions and uses a feature called Role Based Access Control (RBAC) which defines what administrators can and cannot do. When you make a remote PowerShell connection to an Exchange 2016 server, the RBAC authorization module in IIS determines which cmdlets and parameters you have access to. Once this information is obtained, only the cmdlets and parameters that have been assigned to your account via an RBAC role are loaded into your PowerShell session using implicit remoting.

## There's more...

In the previous example, we explicitly set the credentials used to create the remote shell connection. This is optional and not required if the account you are currently logged on with has the appropriate Exchange permissions assigned. To create a remote shell session using your currently logged on credentials, use the following syntax to create the session object:

$session = New-PSSession -ConfigurationName Microsoft.Exchange `

-ConnectionUri http://tlex01.testlabs.se/PowerShell/

Once again, import the session:

Import-PSSession $session

When the tasks have been completed, remove the session:

Remove-PSSession $session

You can see here that the commands are almost identical to the previous example, except this time we've removed the -Credential parameter and used the assigned credential object. After this is done, you can simply import the session and the commands will be imported into your current session using implicit remoting.

In addition to implicit remoting, Exchange 2016 servers running PowerShell v5 or above can also be managed using fan-out remoting. This is accomplished using the   
Invoke-Command cmdlet and it allows you to execute a script block on multiple computers in parallel. For more details, run Get-Help Invoke-Command and   
Get-Help about\_remoting.

Since Exchange Online is commonly used by Microsoft customers nowadays, let’s take a look at an example on how to connect as well, it’s very similar to connecting to remote PowerShell on-premises. The following prerequisites are required; .NET Framework 4.5 or 4.5.1 and then either Windows Management Framework 3.0 or 4.0.

Creating a variable of the credentials:

$UserCredential = Get-Credential

Creating a session variable:

$session = New-PSSession -ConfigurationName Microsoft.Exchange `  
-ConnectionUri https://outlook.office365.com/powershell-liveid/ `  
-Credential $UserCredential -Authentication Basic `  
-AllowRedirection

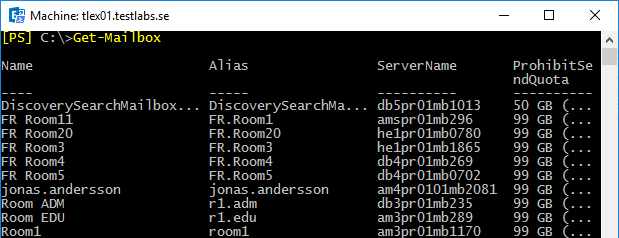
Finally, import the session:

Import-PSSession $session -AllowClobber

Perform the tasks you want to do:

Get-Mailbox

Exchange Online mailboxes are shown in the screenshot below:



7081EN\_02\_03

When the tasks have been completed, remove the session:

Remove-PSSession $session

## See also

Using explicit credentials with PowerShell commands

# Using explicit credentials with PowerShell cmdlets

There are several PowerShell and Exchange Management Shell cmdlets that provide a credential parameter that allows you to use an alternate set of credentials when running a command. You may need to use alternate credentials when making manual remote shell connections, sending e-mail messages, working in cross-forest scenarios, and more. In this recipe, we'll take a look at how you can create a credential object that can be used with commands that support the -Credential parameter.

## How to do it...

To create a credential object, we can use the Get-Credential cmdlet. In this example,   
we store the credential object in a variable that can be used by the Get-Mailbox cmdlet:

$credential = Get-Credential

Get-Mailbox -Credential $credential

## How it works...

When you run the Get-Credential cmdlet, you are presented with a Windows authentication dialog box requesting your username and password. In the previous example, we assigned the Get-Credential cmdlet to the $credential variable. After typing your username and password into the authentication dialog box, the credentials are saved as an object that can then be assigned to the -Credential parameter of a cmdlet. The cmdlet that utilizes the credential object will then run using the credentials of the specified user.

Supplying credentials to a command doesn't have to be an interactive process.   
You can programmatically create a credential object within your script without using   
the Get-Credential cmdlet:

$user = "testlabs\administrator"

$pass = ConvertTo-SecureString -AsPlainText P@ssw0rd01 -Force

$credential = New-Object ` System.Management.Automation.PSCredential `  
-ArgumentList $user,$pass

You can see here that we've created a credential object from scratch without using the   
Get-Credential cmdlet. In order to create a credential object, we need to supply the password as a secure string type. The ConvertTo-SecureString cmdlet can be used to create a secure string object. We then use the New-Object cmdlet to create a credential object specifying the desired user name and password as arguments.

If you need to prompt a user for their credentials but you do not want to invoke the Windows authentication dialog box, you can use this alternative syntax to prompt the user in the shell for their credentials:

$user = Read-Host "Please enter your username"

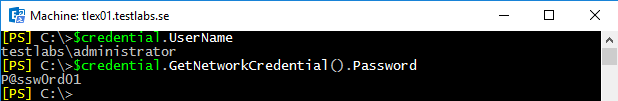
$pass = Read-Host "Please enter your password" -AsSecureString

$credential = New-Object ` System.Management.Automation.PSCredential-ArgumentList ` $user,$pass

This syntax uses the Read-Host cmdlet to prompt the user for both their username   
and password. Notice that when creating the $pass object we use Read-Host with   
the -AsSecureString parameter to ensure that the object is stored as a secure string.

## There's more...

After you've created a credential object, you may need to access the properties of that object to retrieve the username and password. We can access the username and password properties of the $credential object created previously using the following commands:



7081EN\_02\_04

You can see here that we can simply grab the username stored in the object by accessing the UserName property of the credential object. Since the Password property is stored as a secure string, we need to use the GetNetworkCredential method to convert the credential to a NetworkCredential object that exposes the Password property as a simple string.

Another powerful method for managing passwords for scripts is to encrypt them and store into a text file. This can be easily done using the example below.

The password is stored into a variable:

$secureString = Read-Host -AsSecureString "Enter a secret password"

The variable gets converted from SecureString and saved to text file:

$secureString | ConvertFrom-SecureString | Out-File .\storedPassword.txt

The contents in the text file is retrieved and converted into a SecureString value:

$secureString = Get-Content .\storedPassword.txt | ConvertTo-SecureString

## See also

* Transferring files through remote shell connections

Manually Configuring Remote PowerShell Connections

# Transferring files through remote shell connections

Since the Exchange 2016 Management Shell commands are executed through a remote PowerShell session, importing and exporting files requires a new special syntax. There are a handful of shell cmdlets that require this, and in this recipe, we'll take a look at the syntax that needs to be used to transfer files through a remote shell connection.

## How to do it...

Let's say that we are about to import a certificate to the Exchange server, more specifically the client access services. We can import the file using the Get-Content cmdlet, using syntax similar to the following:

[byte[]]$data = Get-Content -Path ".\ExportedCert.pfx" `

-Encoding Byte `

-ReadCount 0

$password = Get-Credential

Import-ExchangeCertificate –FileData $data –Password $password.Password

In this example, the file data is first read into a variable called $data. The certificate import is done by using the Import-ExchangeCertificate cmdlet by assigning the $data variable as a value to the –FileData parameter.

## How it works...

When you launch the Exchange 2016 Management Shell, special commands called proxy functions are imported into your local shell session. These proxy functions represent the compiled cmdlets that are actually installed on your Exchange server. When you run these commands, any data required for input through parameters are transferred through a remote connection from your machine to the server and the command is then executed. Since the commands are actually running on the server and not on your machine, we cannot use a local path for files that need to be imported.

In the previous example, you can see that we first stored the file data in a variable. What   
we are doing here is reading the file content into the variable using the Get-Content  
cmdlet in order to create a byte-encoded object. This variable is then assigned to the   
cmdlet's -FileData parameter, which requires a byte-encoded value.

There are a number of Exchange Management Shell cmdlets that include a -FileData parameter used to provide external files as input:

* Import-DlpPolicyCollection: Used for importing DLP policy collections into the organization
* Import-DlpPolicyTemplate: Used for importing DLP policy template into the organization
* Import-ExchangeCertificate: Used for importing certificates
* Import-JournalRuleCollection: Imports a collection of journal rules
* Import-RecipientDataProperty: Used for importing photos or audio into Active Directory
* Import-TransportRuleCollection: Allows you to import a collection of transport rules
* Import-UMPrompt: Imports custom audio files to UM feature

This is a good example of how remote PowerShell sessions have changed behavior since back in Exchange 2010. For example, if you have worked with the shell in Exchange 2007, you may remember the Import-ExchangeCertificate cmdlet. This cmdlet used to accept a local file path when importing a certificate into a server, but due to the new remoting functionality, the commands used to perform this task have changed, even though the cmdlet name is still the same.

## There's more...

We also have to take remote shell connections into consideration when exporting data. For example, let's say that we need to export the user photo associated with a mailbox from Active Directory. The command would look something like this:

Export-RecipientDataProperty -Identity tdawson-Picture | %{

$\_.FileData | Add-Content C:\pics\tdawson.jpg -Encoding Byte

}

When using the Export-RecipientDataProperty cmdlet with the -Picture switch parameter, the photo can be retrieved from the FileData property of the object returned. The photo data is stored in this property as a byte array. In order to export the data, we need to loop through each element stored in this property and use the Add-Content cmdlet to re-construct the image to an external file.

When dealing with cmdlets that import or export data, make sure you utilize the help system. Remember, you can run Get-Help <cmdlet name> -Examples with any of these cmdlets to determine the correct syntax.

## See also

* Using the Help System in Chapter 1, PowerShell Key Concepts
* Manually Configuring Remote PowerShell Connections
* Importing user photos into Active Directory in Chapter 3, Managing Recipients

# Managing domains or an entire forest using recipient scope

The Exchange Management Tools can be configured to use specific portions of your Active Directory hierarchy using a specific recipient scope. When you set the recipient scope to a location in the Active Directory, such as a domain or an organizational unit, the Exchange Management Shell will only allow you to view the recipients that are stored in that location and any containers beneath it. In this recipe, we'll look at how to set the recipient scope when working with the Exchange Management Shell.

## How to do it...

1. We can set the recipient scope in the Exchange Management Shell using the Set-AdServerSettings cmdlet. For example, to set the recipient scope to the Sales OU in the testlabs.se domain, use the following command:

Set-AdServerSettings -RecipientViewRoot testlabs.se/sales

We can also specify the value using the distinguished name of the OU:

Set-AdServerSettings -RecipientViewRoot `

"OU=sales,DC=testlabs,DC=se"

## How it works...

When you first start the Exchange Management Shell, the default recipient scope is set to the domain of the computer that is running the shell. If you change the recipient scope, the setting will not be retained when you restart the shell. The default domain scope will always be used when you launch the shell. You can override this by adding these commands to your PowerShell profile to ensure that the setting is always initially configured as needed.

In the previous example, we set the recipient scope to a specific OU in the domain. If you are working in a multi-domain forest, you can use the -ViewEntireForest parameter so that all recipient objects in the forest can be managed from your shell session. Use the following command to view the entire forest:

Set-AdServerSettings -ViewEntireForest $true

To change the recipient scope to a specific domain, set the –RecipientViewRoot to the full qualified domain name of the Active Directory domain:

Set-AdServerSettings -RecipientViewRoot corp.testlabs.se

## There's more...

If you're working in a large environment with multiple domains and OUs, setting the recipient scope can improve the speed of the Exchange Management Shell, since it will limit the total number of recipients returned by your commands.

If you have Exchange recipients in multiple Active Directory domains or sites, you may have to take replication latency into account when working with a broad recipient scope. To handle this, you can use the Set-AdServerSettings cmdlet to specify domain controllers and global catalog servers that you want to work with.

To set the preferred domain controllers and global catalog that should be   
used with your recipient scope, use the -SetPreferredDomainControllers  
and –PreferredGlobalCatalog parameters to specify the FQDN of the servers:

Set-AdServerSettings -ViewEntireForest $true `

-SetPreferredDomainControllers dc1.testlabs.se `

-PreferredGlobalCatalog dc1.testlabs.se

Setting the preferred domain controller can be useful to ensure your commands will read the latest list of recipients in Active Directory. If you have a provisioning process that uses a specific domain controller when creating recipients, it may take some time to replicate this information throughout the forest. Setting the preferred domain controllers can be used to ensure that you are working with the latest set of recipients available, even if they haven't been replicated throughout the forest.

# Exporting reports to text and CSV files

One of the added benefits of the Exchange Management Shell is the ability to run very detailed and customizable reports. With the hundreds of Get-\* cmdlets provided between Windows PowerShell and the Exchange Management Shell, the reporting capabilities are almost endless. In this recipe, we'll cover exporting command output to plain text and CSV files that can be used to report on various resources throughout your Exchange environment.

## How to do it...

To export command output to a text file, use the Out-File cmdlet. To generate a report   
of mailboxes in a specific mailbox database that can be stored in a text file, use the   
following command:

Get-Mailbox | Select-Object Name,Alias | Out-File c:\report.txt

You can also save the output of the previous command as a CSV file that can then be opened and formatted in Microsoft Excel:

Get-Mailbox | Select-Object Name,Alias | `

Export-CSV c:\report.csv –NoType

## How it works...

The Out-File cmdlet is simply a redirection command that will export the output of your command to a plain text file. Perhaps one of the most useful features of this cmdlet is the ability to add data to the end of an existing file using the -Append parameter. This allows you to continuously update a text file when processing multiple objects or creating persistent log files or reports.

You can also use the Add-Content, Set-Content, and Clear-Content cmdlets to add, replace, or remove data from files.

The Export-CSV cmdlet converts the object's output, by your command, into a collection of comma-separated values and stores them in a CSV file. When we ran the   
Get-Mailbox cmdlet in the previous example, we filtered the output, selecting only the Name and Alias properties. When exporting, this output using Export-CSV, these property names are used for the column headers. Each object returned by the command will be represented in the CSV file as an individual row, therefore populating the Name and Alias columns with the associated data.

You may have noticed in the Export-CSV example that we used the -NoType switch parameter. This is commonly-used and is shorthand notation for the full parameter   
name -NoTypeInformation. If you do not specify this switch parameter, the first line   
of the CSV file will contain a header specifying the .NET Framework type of the object that was exported. This is rarely useful. If you end up with a strange-looking header in one of your reports, remember to run the command again using the   
–NoTypeInformation switch parameter.

## There's more...

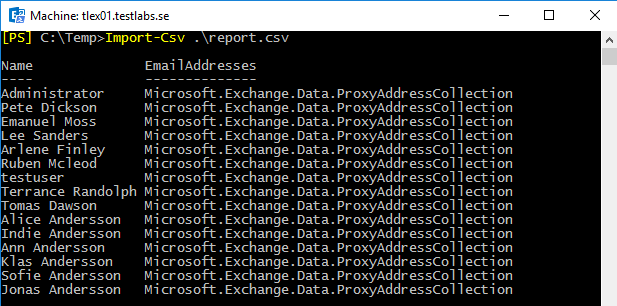
One of the most common problems that Exchange administrators run into with   
Export-CSV is when exporting objects with multi-valued properties. Let's say we need to run a report that lists each mailbox and its associated e-mail addresses. The command would look something like the following:

Get-Mailbox | `

Select-Object Name,EmailAddresses | `

Export-CSV c:\report.csv -NoType

The problem here is that each mailbox can contain multiple e-mail addresses. When we select the EmailAddresses property, a multi-valued object is returned. The Export-CSV cmdlet does not understand how to handle this, and when you import the CSV file in PowerShell, you'll end up with a CSV file that looks like the following:



7081EN\_02\_05

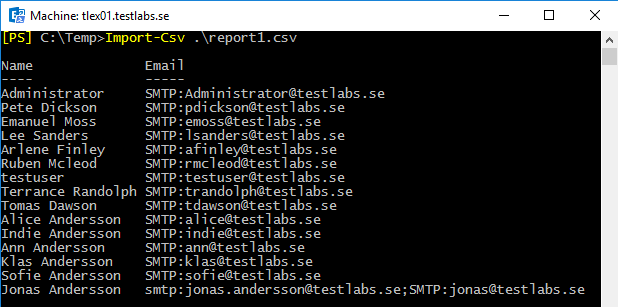
From looking at this screenshot, you can see that on the first line, we have our header names that match the properties selected during the export. In the first column, the Name property for each mailbox has been recorded correctly, but, as you can see, the values listed in the EmailAddresses column have a problem. Instead of the e-mail addresses, we get the .NET Framework type name of the multi-valued property. To get around this, we need to help the Export-CSV cmdlet understand what we are trying to do and specifically reference the data that needs to be exported.

One of the best ways to handle this is to use a calculated property and join each value of the multi-valued property as a single string:

Get-Mailbox | `

Select-Object Name,@{n="Email";e={$\_.EmailAddresses -Join ";"}}` | Export-CSV c:\report1.csv -NoType

In this example, we've modified the previous command by creating a calculated property that will contain each e-mail address for the associated mailbox. Since we need to consolidate the EmailAddresses property data into a single item that can be exported,   
we use the -Join operator to create a string containing a list, separated by semi-colons, of every e-mail address associated with each mailbox. The command is then piped to the Export-CSV cmdlet, and the report is generated in a readable format that can be viewed   
using the Import-CSV cmdlet:



7081EN\_02\_06

As you can see in this screenshot, each e-mail address for a mailbox is now listed in the Email column and is separated using a semi-colon. Each address has an SMTP prefix associated with it. An SMTP prefix in all capital letters indicates that the address is the primary SMTP address for the mailbox. Any remaining secondary addresses use an SMTP prefix in lower case characters. If you do not want to export the prefixes we can modify our code even further:

Get-Mailbox | `

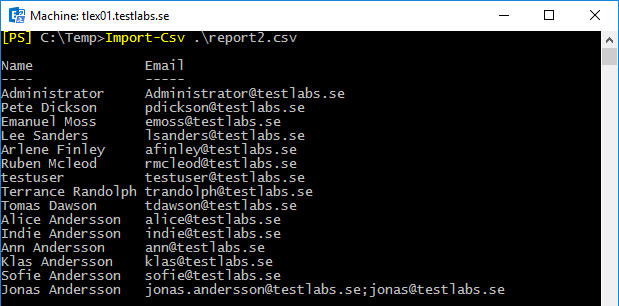
select-Object Name, `

@{n="Email"; `

e={($\_.EmailAddresses | %{$\_.SmtpAddress}) -Join ";"} `

} | Export-CSV c:\report2.csv -NoType

Here you can see that, within the expression of the calculated property, we're looping through the EmailAddresses collection and retrieving only the SmtpAddress, which does not include the SMTP prefix and returns only the e-mail addresses. Once the data is exported to a CSV file we can review it using the Import-CSV cmdlet:



7081EN\_02\_07

As you can see here, we now get each e-mail address associated with each mailbox without the SMTP prefix within the Email column of our CSV file.

## See also

* Working with arrays and hash tables in Chapter 1, PowerShell Key Concepts

Creating custom objects in Chapter 1, PowerShell Key Concepts

# Sending SMTP e-mails through PowerShell

As an Exchange administrator, will probably need an automated solution for sending e-mails from your PowerShell scripts. Whether it's for sending notifications to users in a specific database or e-mailing the output of your scripts to a reporting mailbox, the transmission of messages like these will prove very useful in performing common day-to-day administrative scripting tasks. In this recipe, we'll take a look at how you can send SMTP e-mail messages from PowerShell to the recipients in your Exchange organization.

## How to do it...

PowerShell v2 and later includes a core cmdlet that can be used to send e-mail messages via SMTP to one or more recipients. Use the following syntax to send an e-mail message:

Send-MailMessage -To user1@testlabs.se `

-From administrator@testlabs.se `

-Subject "Test E-mail" `

-Body "This is just a test" `

-SmtpServer tlex01.testlabs.se

## How it works...

In PowerShell v1, the Send-MailMessage cmdlet didn't exist. In the early days before Exchange 2007 SP2 and PowerShell v2 support, we had to use the classes in the System.Net.Mail namespace in the .NET Framework to send SMTP e-mail messages. This was difficult for some administrators because working with .NET classes can be confusing without prior programming experience. The good news is that the Send-MailMessage cmdlet utilizes these same .NET classes that allow you to create rich e-mail messages that can contain one or more attachments, using an HTML formatted message body, support message priority, and more. Here are some of the more useful parameters that can be used with the Send-MailMessage cmdlet:

* Attachments: This specifies the path to the file that should be attached. It separates multiple attachments with a comma.
* Bcc: This allows you to specify a blind-copy recipient. It separates multiple recipients using a comma.
* Body: This specifies the content of a message.
* BodyAsHtml: This is a switch parameter that ensures the message will use an HTML-formatted message body.
* Cc: This allows you to specify a carbon-copy recipient. It separates multiple recipients using a comma.
* Credential: You can provide a PSCredential object created by the Get-Credential cmdlet to send the message using the credentials of another user.
* DeliveryNotificationOption: This specifies the delivery notification options for the message. The default value is None, but other valid options are OnSuccess, OnFailure, Delay, and Never.
* Encoding: This specifies the encoding of the email, such as S/MIME and non-MIME character set.
* From: This is the e-mail address of the sender. You can define a display name using the following format: Dave <dave@contoso.com>.
* Priority: This specifies the importance of the message. The default value is Normal. The remaining valid values are High and Low.
* SmtpServer: This needs to be the name or IP address of your SMTP server. When working in an Exchange environment, this will be set to one of your Hub Transport servers.
* Subject: This is the subject of the e-mail message.

To: This allows you to specify an e-mail recipient. It separates multiple recipients with a comma.

UseSsl: This allows you to specify that the connection should be established using SSL.

Port: This allows you to specify an alternate port for the SMTP server.

## There's more...

When using this cmdlet, you'll need to specify an SMTP server in order to submit the message. Unless you are already using some type of mail relay system within your environment, you'll want to use a Mailbox server in your Exchange organization. Out of the box, Exchange servers will not allow workstations or untrusted servers to relay e-mail messages. Depending on where you are sending the message from, you may need to allow the machine running your scripts to relay e-mail.

PowerShell v2 and later includes a preference variable called $PSEmailServer that can be assigned the name or IP address of an SMTP server. When this variable is defined, you can omit the -SmtpServer parameter when using the Send-MailMessage cmdlet. You can add this variable assignment to your PowerShell profile so that the setting will persist across all of your shell sessions.

### Sending messages with attachments

You may want to write a script that generates a report to a text or CSV file and then e-mail that data to an administrator mailbox. The –Attachment parameter can be used with the Send-MailMessage cmdlet to do this. For example, let's say you've generated a CSV report file for the top 10 largest mailboxes in your environment and it needs to be e-mailed to your staff. The following command syntax could be used in this scenario:

Send-MailMessage -To support@testlabs.se `

-From powershell@testlabs.se `

-Subject "Mailbox Report for $((Get-Date).ToShortDateString())" `

-Body "Please review the attached mailbox report." `

-Attachments c:\report.csv `

-SmtpServer tlex01.testlabs.se

Notice that all we need to do here is provide the path and file name to the -Attachment parameter. You can send multiple message attachments this way by providing a   
comma-separated list of files.

### Sending command output in the body of a message

Instead of exporting command data to an external file and sending it as an attachment, you may want to add this information to the body of an e-mail. In this example, we'll send a message that displays the top 10 largest mailboxes in the organization in the body of an HTML-formatted message:

[string]$report = Get-MailboxDatabase |

Get-MailboxStatistics| ?{(!$\_.DisconnectDate) -and ` ($\_.DisplayName -notlike "\*HealthMailbox\*")} |

Sort-Object TotalItemSize -Desc |

Select-Object DisplayName,Database,TotalItemSize -First 10 |

ConvertTo-Html

Send-MailMessage -To support@testlabs.se `

-From powershell@testlabs.se `

-Subject "Mailbox Report for $((Get-Date).ToShortDateString())" `

-Body $report `

-BodyAsHtml `

-SmtpServer tlex01.testlabs.se

Here you can see that the report data is generated with a fairly sophisticated one-liner and the output is saved in a string variable called $report. We need to strongly type the $report variable as string because that is the data type required by the -Body parameter of the Send-MailMessage cmdlet. Notice that we're using the ConvertTo-Html cmdlet at the end of the one-liner to convert the objects to an HTML document. Since the $report variable will simply contain raw HTML, we can assign this value to the –Body parameter and use the -BodyAsHtml switch parameter to send the report data in the body of an HTML-formatted message.

## See also

* Allowing application servers to relay mail in Chapter 8, Managing Transport Servers
* Sending e-mail messages with EWS in Chapter 14, Scripting with the Exchange Web Services Managed API

Reporting on mailbox size in Chapter 4, Managing Mailboxes

# Scheduling scripts to run at a later time

One of the most common tasks that Exchange administrators perform is scheduling scripts to run at a later time. This can be useful when performing maintenance after hours or running monitoring scripts on a regular basis. In this recipe, you'll learn how to schedule your PowerShell scripts to run with the Windows Task Scheduler.   
In PowerShell version 4 we got some powerful new cmdlets for managing Windows Task Scheduler.

1. Open the Exchange Management Shell by clicking the windows button and go to Microsoft Exchange Server 2016 | Exchange Management Shell.

## How to do it...

To create a scheduled task that runs from one of your Exchange servers use the   
following steps:

1. Open the Task Scheduler by clicking the windows button and on Windows Administrative Tools | Task Scheduler.
2. From the Task Scheduler Library | Action menu, click Create Basic Task.
3. Give your task a name and description, and click Next.
4. On the Trigger screen, select the how often you'd like the script to run (Daily, Weekly, Monthly, and so on).
5. When asked what action you want the task to perform, select Start a Program.
6. Use the following syntax in the Program/Script field and click on Next:   
   C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -command ". 'C:\Program Files\Microsoft\Exchange Server\V15\bin\RemoteExchange.ps1'; Connect-ExchangeServer -auto; c:\Scripts\MoveMailboxes.ps1".
7. You will receive a prompt that says It appears as though arguments have been included in the program text box. Do you want to run the following program?  
   Click Yes.

This will bring you to a summary screen where you can click Finish.

## How it works...

The syntax used in this example may look a little strange at first. What we are actually doing here is scheduling PowerShell.exe and using the -Command parameter to execute multiple statements. This allows us to pass the contents of a PowerShell script to PowerShell.exe. In this case, our script has multiple lines and each statement is separated by a semi-colon.

The first thing we do is dot-source the RemoteExchange.ps1 script located in the Exchange Server bin directory. This file initializes some Exchange shell variables and imports several Exchange specific functions.

The next line of in the task calls the Connect-ExchangeServer function using the   
-Auto parameter, allowing the Exchange Management Shell environment to load automatically from the best Exchange Server in the local AD site.

Finally, we provide the path to our .ps1 script that utilizes any required Exchange Management Shell cmdlets and the script is executed, carrying out whatever it is that we need to be done.

It's worth mentioning here that you do not have to use a .ps1 script file with this syntax. You can replace the call to the MoveMailboxes.ps1 file with any valid PowerShell commands. If you have a script that contains multiple lines, you can continue to separate each line using a semi-colon.

When using this method, make sure that you configure the scheduled task to run as a user that has administrative access to your Exchange organization. In addition, RBAC should be considered for minimizing and using the least required privileges when dealing with accounts that are used for running actions within the task scheduler.

Also, if you have User Account Control (UAC) enabled, you may need to enable the option to Run with highest privileges in the properties of the scheduled task, this for using elevated privileges. Additionally, you will probably want to enable the option to Run whether user is logged on or not in the properties of the scheduled task.

## There's more...

The previous example demonstrated scheduling a task from an Exchange server using the installed Exchange Management Shell tools. Since all of the Exchange Management Shell connections utilize PowerShell remoting, it is possible to schedule a script to run from a workstation or server without the Exchange tools installed.

To schedule a task from a machine without the Exchange tools installed, use the steps from the previous example, but use the following syntax for the program action:

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -command "$s = New-PSSession -ConfigurationNameMicrosoft.Exchange -ConnectionUri http://tlex01.contoso.com/PowerShell/; Import-PSSession $s ; c:\Scripts\MoveMailboxes.ps1"

You can see here again we are scheduling the PowerShell.exe program and specifying the script using the -Command parameter. The difference is that this time we are not using the locally installed Exchange tools. Instead we are creating a manual implicit remoting connection to a particular Exchange server. The length of the command line wrapping makes it difficult to read, but keep in mind that this is all done on one line.

When using this method, you can configure the scheduled task to run as a user that has administrative access to your Exchange organization, or you can provide explicit credentials used to create the session object and run the script as another user.

Scheduled tasks can since version 4 of PowerShell also be added using the cmdlets.

An example of that would look like below.

$TaskCommand = ` "c:\windows\system32\WindowsPowerShell\v1.0\powershell.exe"

$TaskArg = '-command "$s = New-PSSession -ConfigurationName ` Microsoft.Exchange -ConnectionUri ` http://tlex01.testlabs.se/PowerShell/; Import-PSSession $s; ` c:\Scripts\MoveMailboxes.ps1"'

$TaskStartTime = [datetime]::Now.AddMinutes(15)

$TaskAction = New-ScheduledTaskAction -Execute "$TaskCommand" `  
-Argument "$TaskArg"

$TaskTrigger = New-ScheduledTaskTrigger -At $TaskStartTime -Once

Register-ScheduledTask -Action $TaskAction -Trigger $Tasktrigger ` -TaskName "Scheduled task - Move Mailboxes" –User ` "testlabs\administrator" -RunLevel Highest

We are created a variable named TaskCommand, which refers to powershell.exe including the full path. Second row we are creating the variable TaskArg, it’s used for what arguments to powershell.exe should be used. TaskStartTime is using the current time, plus 15 minutes ahead. Finally, we are using these variables and registering the scheduled task.

## See also

* Manually configuring remote PowerShell connections
* Using explicit credentials with PowerShell cmdlets

# Logging shell sessions to a transcript

You may find it useful at times to record the output of your shell sessions in a log file. This can help you save the history of all the commands you've executed and determine the success or failure of automated scripts. In this recipe, you'll learn how to create a PowerShell transcript. An example when this could be useful, if we are developing a script and about to implement it in a production environment. It would be neat to use transcript logging during the first run(s).

## How to do it...

1. To create a transcript, execute the Start-Transcript cmdlet:

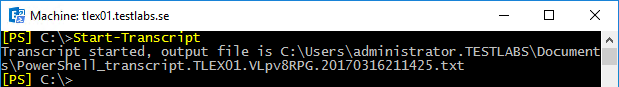
Start-Transcript c:\logfile.txt

2. You can stop recording the session using the Stop-Transcript cmdlet:

Stop-Transcript

## How it works...

When starting a PowerShell transcript, you can specify a path and a file name that will be used to record your commands and their output. The use of the -Path parameter is optional; if you do not provide a file path, the cmdlet will create a transcript file with a random name in the default documents folder in your profile path, as shown in the following screenshot:



7081EN\_02\_08

When you are done, you can run the Stop-Transcript cmdlet or simply exit the shell. You can use the -Append parameter with the Start-Transcript cmdlet to add a new transcript to an existing log file. When doing so, you'll need to specify the name of the file you want to append to using the -Path parameter.

You can record your entire session every time you start the Exchange Management Shell by adding the Start-Transcript cmdlet to your user profile. If you choose to do this, make sure you specify the same log file to use every time the shell starts and use the   
-Append parameter so that each session is added to the log file every time. Note that the transcript file could be huge over time when logging everything.

## There's more...

By default, only the output from PowerShell cmdlets will be recorded in your transcript. If you execute an external program, such as the Exchange eseutil.exe utility, the output from this command will not be saved in your transcript file, even though it was run within the current shell session. You can pipe external programs to the Out-Default cmdlet and this will force the output to be stored in your transcript.

## See also

Exporting reports to text and CSV files

Automating tasks with the scripting agent

# Automating tasks with the scripting agent

The scripting agents were introduced in Exchange 2010 and still remains in Exchange 2016.The concept of cmdlet extension agents is to extend the functionality of the Exchange Management Tools. The scripting extension agent can be used to trigger custom commands as changes are being made by administrators from the management console or the shell. In this recipe, we'll take a look at how to use the scripting agent to automate a task in the Exchange Management Shell.

## Getting ready

To complete the steps in the recipe, you'll need to create an XML file. You can simply use Notepad or any XML editor of your choice.

## How to do it...

1. Let's say that you need to enable single item recovery for every mailbox that gets created in your organization. By default, single item recovery is disabled when you create a mailbox. To automatically enable single item recovery for each mailbox as it is created, add the following code to a new file:

<?xml version="1.0" encoding="utf-8" ?>

<Configuration version="1.0">

<Feature Name="MailboxProvisioning" Cmdlets="New-Mailbox">

<ApiCall Name="OnComplete">

if($succeeded) {

$mailbox =   
 $provisioningHandler.UserSpecifiedParameters["Name"]

Set-Mailbox $mailbox -SingleItemRecoveryEnabled $true

}

</ApiCall>

</Feature>

</Configuration>

1. Next, make sure to save the file as ScriptingAgentConfig.xml on all Exchange servers in the <install path>\V15\Bin\CmdletExtensionAgents directory.

Finally, you need to enable the scripting agent using the following command:

Enable-CmdletExtensionAgent "Scripting Agent"

If you have multiple Exchange servers in your environment, make sure that you copy the ScriptingAgentConfig.xml file to each server into the CmdletExtentionAgents directory as described previously.

## How it works...

When the scripting agent is enabled, it is called every time a cmdlet is run in your Exchange environment. This includes cmdlets run from within the shell or any of the graphical management tools.

You can see from the code that, in this example, we're using the OnComplete API, which   
runs immediately after the cmdlet has been completed. Using the Feature tag, we've specified that this block of code should only be executed upon completion of the   
New-Mailbox cmdlet.

After the New-Mailbox cmdlet has completed, we check the built-in $succeeded variable to ensure the command was successful. If so, we retrieve the value that was used with the -Name parameter and store the result in the $mailbox variable. This value is then used to specify the identity when running the Set-Mailbox cmdlet to enable single item recovery.

## There's more...

You can add multiple scripts to the XML file if needed by defining multiple Feature tags under the configuration tag. Each block of code within the Feature tag should have an ApiCall tag as shown in the previous example.

The state of the scripting agent is an organization-wide setting. If you enable the scripting agent, it is important that the ScriptingAgentConfig.xml is copied to every Exchange server in your organization.

### Using multiple cmdlets with the OnComplete API

Let's take a look at another example. Imagine that, in addition to enabling single-item   
recovery for all newly-created mailboxes, we also want to disable the ActiveSync protocol for each mailbox. This means that, in addition to calling the Set-Mailbox cmdlet to enable single item recovery, we'll also need to call the Set-CASMailbox cmdlet to disable ActiveSync. Also, mailboxes can be created using both the New-Mailbox and Enable-Mailbox cmdlets. Since we'd like our custom settings to be applied regardless of how the mailbox is created, we can use the following code in our XML file:

<?xml version="1.0" encoding="utf-8" ?>

<Configuration version="1.0">

<Feature Name="Mailboxes" Cmdlets="new-mailbox,enable-mailbox">

<ApiCall Name="OnComplete">

if($succeeded) {

$id = $provisioningHandler.UserSpecifiedParameters["Name"]

Set-Mailbox $id -SingleItemRecoveryEnabled $true

Set-CASMailbox $id -ActiveSyncEnabled $false

}

</ApiCall>

</Feature>

</Configuration>

This code is similar to our previous example, except in this version we've specified that our custom code will be called when both the New-Mailbox and Enable-Mailbox cmdlets are used. The code in the ApiCall tag captures the Alias of the mailbox and then uses the Set-Mailbox and Set-CASMailbox to modify the settings as required.

There are multiple scripting agent APIs that can be used to extend the Exchange Management Shell functionality even further. For examples on how to use these APIs, reference the ScriptingAgentConfig.xml.sample file in the <installpath>\V15\Bin\CmdletExtensionAgents folder.

## See also

* Adding, modifying, and removing mailboxes in Chapter3, Managing Recipients

Managing ActiveSync, OWA, POP3, and IMAP4 mailbox settings in Chapter 7, Managing Client Access

# Scripting an Exchange server Installation

If you are performing mass deployment of Exchange servers in a large environment, automating the installation process can minimize administrator error and speed up the   
overall process. The setup.exe utility can be used to perform an unattended installation   
of Exchange, and, when combined with PowerShell and just a little bit of scripting logic,   
create a fairly sophisticated installation script. This recipe will provide a couple of examples that can be used to script the installation of an Exchange server.

## Getting ready

You can use a standard PowerShell console from the server to run the scripts in this recipe.

## How to do it...

1. In this example, we'll create an automated installation script that installs Exchange based on the host name of the server. Using Notepad or your favourite scripting editor, add the following code to a new file:

Param($Path)

if(Test-Path $Path) {

switch -wildcard ($env:computername) {

"\*-MB-\*" {$role = "MB" ; break}

"\*-MG-\*" {$role = "MT" ; break}

"\*-ED-\*" {$role = "ET" ; break}

}

$setup = Join-Path $Path "setup.exe"

Invoke-Expression "$setup /mode:install `  
/r:$role /IAcceptExchangeServerLicenseTerms ` /InstallWindowsComponents"

}

else {

Write-Host "Invalid Media Path!"

}

1. Save the file as InstallExchange.ps1.

Execute the script from a server where you want to install Exchange using the following syntax:

InstallExchange.ps1 -Path D:

The value provided for the -Path parameter should reference the Exchange 2016 media, either on DVD or extracted to a folder.

## How it works...

One of the most common methods for automating an Exchange installation is determining the required roles based on the hostname of the server. In the previous example, we assume that your organization uses a standard server naming convention. When executing the script, the switch statement will evaluate the hostname of the server and determine the required roles. For example, if your mailbox servers use a server name such as TL-MB-01, the mailbox server role(s) will be installed. If your Management server(s) use a server name such as TL-MG-01, the Management role will be installed, and so on.

It's important to note that Exchange 2016 requires several Windows operating system roles and features, such as .NET Framework 4.6. This of course is pending on which role we are installing, the required roles and features will be installed by using the switch /InstallWindowsComponents.

When calling the Setup.exe installation program within the script, we use the /InstallWindowsComponents and /IAcceptExchangeServerLicenseTerms switch, which is new Setup.exe features in Exchange Server 2016. This will allow the setup program to load any prerequisite Windows roles and features, such as IIS, and so on, before starting the Exchange installation. The accept agreement switch is required when using the unattended installation method.

## There's more...

Scripting the installation of Exchange based on the server names may not be an option for you. Fortunately, PowerShell gives us plenty of flexibility. The following script uses similar logic, but performs the installation based on different criteria.

Let's say that your core Exchange infrastructure has already been deployed. Your corporate headquarters already has the required Mailbox server infrastructure in place and therefore you only need to deploy mailbox servers in the Active Directory branch sites. All remaining remote sites will also be installing the Mailbox role for the Exchange servers. Replace the code in the InstallExchange.ps1 script with the following:

param($Path)

$site = [DirectoryServices.ActiveDirectory.ActiveDirectorySite]

if(Test-Path $Path) {

switch ($site::GetComputerSite().Name) {

"EX-Deploy" {$role = "MB"}

Default {$role = "MT"}

}

$setup = Join-Path $Path "setup.exe"

Invoke-Expression "$setup /mode:install /r:$role /IAcceptExchangeServerLicenseTerms /InstallWindowsComponents"

}

else {

Write-Host "Invalid Media Path!"

}

This example above determines the current Active Directory site of the computer executing the script. If the computer is in the EX-Deploy site, meaning that the server is placed in the deployment AD site, by having the server in this site the Mailbox role will be installed. If it is located at any of the other remaining Active Directory sites, the Management server role is installed.

As you can see, combining the Setup.exe utility with a PowerShell script can give you many more options when performing an automated installation.

## See also

* Looping through items in Chapter 1, PowerShell Key Concepts

Creating custom objects in Chapter 1, PowerShell Key Concepts