14

Scripting with the Exchange Web Services Managed API

In this chapter, we will cover the following:

* Getting connected to EWS
* Sending e-mail messages with EWS
* Working with impersonation
* Searching mailboxes
* Retrieving the headers of an e-mail message
* Deleting e-mail items from a mailbox
* Creating calendar items

Exporting attachments from a mailbox

# Introduction

Exchange Web Services (EWS) is available for Exchange 2016 (also for Exchange Online). During the writing of the book we’ve used EWS Managed API version 2.2 since that was the latest available release. The API gives developers the ability to write applications that previously used legacy API’s.

Today, developers can call Exchange Management Shell cmdlets from .NET-managed applications to perform administrative tasks programmatically. When it comes to manipulating the contents of a mailbox, such as creating or modifying calendar items, e-mail messages, contacts, or tasks, developers now use EWS.

Working with EWS requires formatting and sending an XML request over HTTP and parsing the XML response from an Exchange server. Initially, developers used either raw XML or auto-generated proxy classes in Visual Studio to do this, and it required some very verbose code that was difficult to read and debug. Fortunately, the Exchange Web Services team developed and released the EWS Managed API in April of 2009. The EWS Managed API is a fully object-oriented .NET wrapper for the EWS XML protocol that makes life much easier for application developers.

Applications written using the Managed API require a fraction of the code that developers had to write previously when working with raw XML or auto-generated proxy classes. This makes for a huge increase in productivity because the code is easier to read and troubleshoot, and the learning curve for new developers is much lower. The good news is that this is also true for Exchange administrators that want to write advanced PowerShell scripts that utilize EWS. The EWS Managed API can be used to do things in PowerShell that are not possible with Exchange Management Shell cmdlets. The EWS Managed API assembly can be loaded into the shell, and, with the right permissions, you can immediately start building scripts that can access and manipulate the data within any mailbox inside the organization.

In this chapter, we will cover some of the key concepts of using EWS in your PowerShell scripts, such as connecting to EWS, sending e-mail messages, and working with items in one or more mailboxes. The end goal is to give you a basic understanding of the EWS Managed API so that you can start building some basic scripts or deciphering the code samples you come across on the internet or within the TechNet documentation.

## Performing some basic steps

To work with the code samples in this chapter, follow these steps to download the EWS Managed API:

1. Download the EWS Managed API from the following URL:  
   http://www.microsoft.com/en-us/download/details.aspx?id=42951
2. Download and run EwsManagedApi.msi.
3. During the installation, select a destination folder such as C:\EWS or choose the default directory C:\Program Files\Microsoft\Exchange\Web Services\2.2. You will need to note the location so you can import the Microsoft.Exchange.WebServices.dll assembly into the shell.

You can use either a standard PowerShell console or the Exchange Management Shell to run the code for each recipe in this chapter.

# Getting connected to EWS

When working with EWS, you first need to create an instance of the ExchangeService class that can be used to send SOAP messages to an Exchange server. This class has several properties and methods that can be used to specify explicit credentials, set the web service's end-point URL, or make a connection using the built-in AutoDiscover client. In this recipe, you'll learn how to make a connection to EWS that can be used to run custom scripts against the web service.

## How to do it...

1. The first thing we need to do is load the EWS Managed API assembly into the shell:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

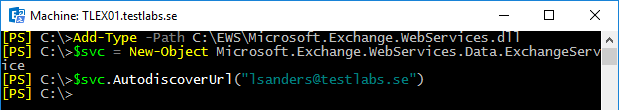
2. Now we can create an instance of the ExchangeService class:

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

3. At this point, we can use the AutoDiscoverUrl method to determine the EWS end-point on the closest Client Access Server for the mailbox with a particular SMTP address:

$svc.AutoDiscoverUrl("lsanders@testlabs.se")

Now that we have an Exchange service connection created, we can send e-mail messages, create and modify items within a mailbox, and perform other tasks. The outcome will look similar to the screenshot:



7081EN\_14\_01

## How it works...

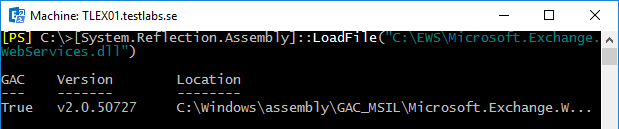
Before we can start working with the classes in the EWS Managed API, the assembly must be loaded so that the .NET Framework types are available when running scripts that utilize the API. This is only valid for the current shell session, and, if you will be creating scripts, you'll want to make sure that this is always the first thing that is done before invoking any code. We used the Add-Type cmdlet in the previous example to load the assembly, but this is also valid:

[System.Reflection.Assembly]::LoadFile(

"C:\ews\Microsoft.Exchange.WebServices.dll"

)

This is basically the longhand method of doing the same thing we did before: loading an unreferenced assembly into the shell environment. Notice that in both examples, we are   
using the path C:\EWS. This is not the default path where the assembly is installed, but you can copy it to any folder of your choice. See screenshot below for example:



7081EN\_14\_02

When creating an instance of the ExchangeService class, we have the option of versioning the connection. For example:

$svc=New-Object Microsoft.Exchange.WebServices.Data.ExchangeService `

-ArgumentList "Exchange2013\_SP1"

Here we are passing the Exchange version to the ExchangeService class constructor. When you do not provide a value, the most recent version of Exchange will be used, which in this case would be Exchange 2013 SP1, since were using the 2.2 version of the API. The values for that can be used for Exchange are Exchange2007\_SP1, Exchange2010, Exchange2010\_SP1, Exchange2010\_SP2, Exchange2013 and Exchange2013\_SP1.

Since we didn't specify credentials when creating the ExchangeService object, we need to provide the SMTP address associated with the mailbox of the currently logged on user when calling the AutoDiscoverUrl method.

## There's more...

If you want to use explicit credentials when creating your ExchangeService object rather than using the credentials of the currently logged on user, you need to do a couple of things differently. The following code will create an instance of the ExchangeService class using an alternate set of credentials:

[System.Reflection.Assembly]::LoadFile(

"C:\ews\Microsoft.Exchange.WebServices.dll"

)

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.Credentials = New-Object `

Microsoft.Exchange.WebServices.Data.WebCredentials `

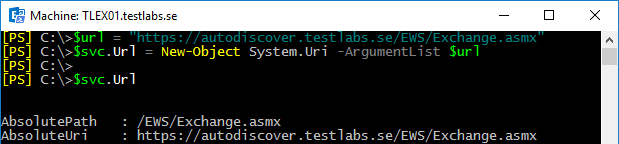
-ArgumentList "jonand","P@ssw0rd01","testlabs.se"

In addition, you also have the option of setting the EWS URL manually:

$url = "https://autodiscover.testlabs.se/EWS/Exchange.asmx"

$svc.Url = New-Object System.Uri -ArgumentList $url

Although it is possible to set the URL manually, developers use AutoDiscover as a best practice because it allows the API to determine the best Client Access Server that should be used as the web service's URL. A hard-coded URL value could potentially mean a broken script if things change later on in your environment. The outcome of the cmdlets above will look similar to screenshot below:



7081EN\_14\_03

## Certificates matter

Just like Outlook Web App, the EWS virtual directory is secured with an SSL certificate. If you are still using the self-signed certificates that are installed by default on Client Access Servers, you'll need to override a security check done by the API to validate the certificate, otherwise you will be unable to connect. To do this, we can use the ServicePointManager class in the System.Net namespace. This class can be used to hook up a certificate validation callback method, and, as long as that method returns $true, the API will consider the self-signed certificate to be trusted:

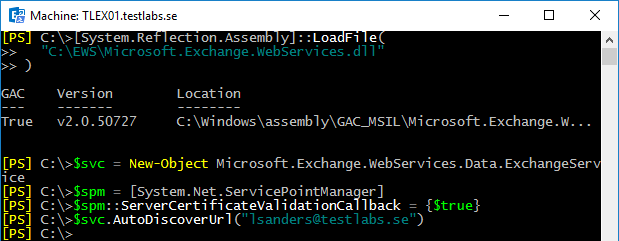
$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$spm = [System.Net.ServicePointManager]

$spm::ServerCertificateValidationCallback = {$true}

$svc.AutoDiscoverUrl("lsanders@testlabs.se")

Certificate validation callback methods are written to perform additional checks on a certificate. These callback methods return a Boolean value that indicates whether or not   
a certificate can be trusted. Instead of writing a callback method, we're assigning a script block that returns $true to the ServerCertificateValidationCallback property. This forces the API to consider any EWS end-point to be secure, regardless of the status of the certificate used to secure it. Keep in mind, that self-signed certificates are considered to be a bootstrap security configuration so connections to Exchange can be secured out of the box. The best practice is to replace these certificates with trusted commercial or enterprise PKI certificates. For example see the screenshot below regarding the URL override:



7081EN\_14\_04

# Sending e-mail messages with EWS

As we saw back in Chapter 2, Exchange Management Shell Common Tasks, we can use the built-in PowerShell cmdlet Send-MailMessage to send e-mail messages. This can be a useful tool when writing scripts that need to send notifications, but the EWS Managed API has several distinct advantages over this approach. In this recipe, we'll take a look at how to send e-mail messages through EWS and why this might be a better option for organizations that have an Exchange infrastructure in place.

## How to do it...

1. First, we'll import the EWS Managed API assembly, create an instance of the ExchangeService class, and set the EWS end-point using AutoDiscover:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("administrator@contoso.com")

2. Next, we'll create an instance of the EmailMessage class:

$msg = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.EmailMessage `

-ArgumentList $svc

3. At this point, we can set specific properties on the $msg object such as the subject, body, and one or more recipients:

$msg.Subject = "Test E-Mail"

$msg.Body = "This is a test"

$msg.From = "administrator@contoso.com"

$msg.ToRecipients.Add("sysadmin@contoso.com")

$msg.SendAndSaveCopy()

Once this code has been executed, the message is sent to sysadmin@contoso.com.

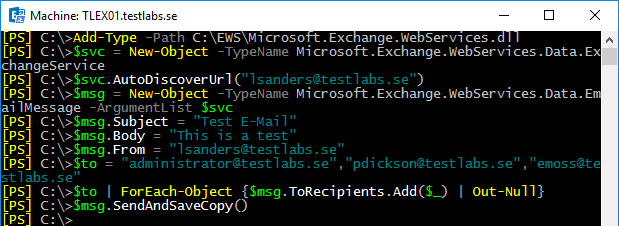
## How it works...

When we send e-mail messages through EWS, we don't have to worry about specifying an SMTP server since the message is transmitted through the web service. This allows our code to run on any machine that has PowerShell installed, and we don't need to modify the receive connectors on the client access servers to allow a specific host to relay mail. Additionally, EWS will allow us to use AutoDiscover to automatically find the correct end-point, which prevents the need to hardcode server names into our scripts.

Setting the Subject, Body, and From properties of an EmailMessage object is pretty straightforward. We simply need to assign a value as we would with any other object. Adding recipients requires that we use the Add method of the ToRecipients property. If you have multiple recipients that must be addressed, you can call this method for each one, or you can loop through a collection using the ForEach-Object cmdlet:

$to = "sysadmin@contoso.com","IT@contoso.com","help@contoso.com"  
$to | ForEach-Object {$msg.ToRecipients.Add($\_)}

When you call the Add method, you'll notice that the ToRecipients property will be returned for each address added to the message. For example, see screenshot below, maybe you did notice that I have used the command Out-Null, it is used for having a proper screenshot size.



7081EN\_14\_05

If you want to simply call this method without having anything returned to the screen, pipe the command to Out-Null (like the screenshot above):

$msg.ToRecipients.Add("sales@contoso.com") | Out-Null

In addition, we can also carbon copy and blind copy recipients on the message:

$msg.CcRecipients.Add("sales@contoso.com") | Out-Null

$msg.BccRecipients.Add("dmsith@contoso.com") | Out-Null

Finally, if you do not want to save a copy of the message in the Sent Items folder, you can simply use the Send method:

$msg.Send()

Keep in mind that, since we did not provide credentials when connecting to EWS, the user running this code will need to have a mailbox on the server which corresponds to the From address being used. Since we are connecting with our currently logged on credentials, the message must be sent from the mailbox of the user running the code.

## There's more...

Instead of typing all of the commands required to instantiate the Exchange service   
object, it makes much more sense to put this code into a reusable function. Call   
AutoDiscover, create the e-mail message object, and set all of the required properties, Consider the following example:

function Send-EWSMailMessage {

param(

[Parameter(

Position=0,

Mandatory=$true,

ValueFromPipelineByPropertyName=$true

)]

[String[]]

$PrimarySmtpAddress,

[Parameter(

Position=1, Mandatory=$true

)]

[String]

$From,

[Parameter(

Position=2, Mandatory=$true

)]

[String]

$Subject,

[Parameter(

Position=3, Mandatory=$true

)]

[String]

$Body,

[Parameter(

Position=4, Mandatory=$false

)]

[String[]]

$Cc,

[Parameter(

Position=5, Mandatory=$false

)]

[String[]]

$Bcc

)

begin {

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

}

process {

$svc = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutodiscoverUrl($From)

$msg = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.EmailMessage `

-ArgumentList $svc

$msg.Subject = $Subject

$msg.Body = $Body

$PrimarySmtpAddress | %{

$msg.ToRecipients.Add($\_) | Out-Null

}

if ($Cc –ne $null)

{

$msg.CcRecipients.Add($Cc) | Out-Null

}

if ($Bcc –ne $null)

{

$msg.BccRecipients.Add($Bcc) | Out-Null

}

$msg.SendAndSaveCopy()

}

}

This is an advanced function that can be run in a couple of different ways. Notice that the first parameter is called PrimarySmtpAddress and it accepts a value from the pipeline by property name. This will allow us to add the function to the Exchange Management Shell and take advantage of the pipeline to send e-mail messages. For example, once this function has been loaded into EMS, we can do something like this:

Get-Mailbox -OrganizationalUnit contoso.com/sales |

Send-EWSMailMessage -From administrator@contoso.com `

-Subject 'Sales Meeting' `

-Body 'Tomorrows sales meeting has been cancelled' `  
 -Cc administrator@contoso.com `  
 -Bcc manager@contoso.com

Here, you can see that we're retrieving all the users from the Sales OU and piping those objects to our Send-EWSMailMessage function. One message will be addressed and sent to each recipient because the PrimarySmtpAddress parameter receives its value from each object that comes across the pipeline. Notice that a carbon copy email will be sent to the specified mailbox when using the Cc parameter and a blind carbon copy email will be sent to the specified mailbox using the Bcc parameter.

Since the PrimarySmtpAddress parameter also accepts an array of string objects, we can run the function and specify a list of recipients, as shown in the following example:

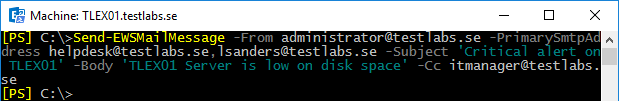
Send-EWSMailMessage -From administrator@testlabs.se `

-PrimarySmtpAddress helpdesk@testlabs.se,lsanders@testlabs.se `

-Subject 'Critical alert on TLEX01' `

-Body 'TLEX01 Server is low on disk space' `  
-Cc itmanager@testlabs.se

Also notice that the email will be sent as a carbon copy to the specified mailbox using the Cc parameter. This could be helpful if the function is used for sending out important emails that might be of interest to the manager too. Both the Cc and Bcc parameters can be used and accepts an array of string objects. The screenshot below shows an example on how the PowerShell function can be used:



7081EN\_14\_06

## See also

Sending SMTP e-mails through PowerShell in Chapter 2, Exchange Management Shell Common Tasks

# Working with impersonation

When building PowerShell scripts that leverage the EWS Managed API, we can use impersonation to access a user's mailbox on their behalf without having to provide their credentials. In order to utilize impersonation, we need permissions inside the Exchange organization, and then we need to configure the ExchangeService connection object   
with the impersonated user ID. In this recipe, you'll learn how to assign the permissions   
and write a script that uses EWS impersonation.

## Getting ready

You will need to use the Exchange Management Shell in this recipe in order to assign permissions for Application Impersonation.

## How to do it...

The first thing you need to do is assign your account the ApplicationImpersonation RBAC role from the Exchange Management Shell:

New-ManagementRoleAssignment -Role ApplicationImpersonation `

-User administrator

After we've been granted the permissions, we need to import the EWS Managed API assembly and configure the ExchangeService connection object:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress","dsmith@contoso.com"

$svc.ImpersonatedUserId = $id

$svc.AutoDiscoverUrl("dsmith@contoso.com")

We now have an ExchangeService connection to EWS as the impersonated user dsmith.

## How it works...

In order to access a mailbox using the permissions of an impersonated user, we use RBAC to create a management role assignment for the user that will be calling the code. Like any other management role assignment, this can be done directly for one user or to a group. Keep in mind that you can also associate scopes when assigning the ApplicationImpersonation role. The command shown in our example would give the administrator account impersonation rights to any mailbox in the organization.

Once we have impersonation rights, we load the EWS Managed API assembly and create an instance of the ExchangeService class to bind to the EWS endpoint.

Notice that, when we create the $id object, we're creating an instance of the ImpersonatedUserId class and passing two values to the constructor. First, we specify that we want to identify the user to impersonate, using a data type of SmtpAddress. The next value passed to the constructor is the actual e-mail address for the impersonated user. The final step is to assign this object to the $svc.ImpersonatedUserId property.

Now that our ExchangeService connection is configured for impersonation, we can   
do things like send e-mails, modify calendar items, or search the mailbox of the   
impersonated user.

## There's more...

Let's take a look at how we could use impersonation using a modified version of the   
Send-EwsMailMessage function, included in the Sending e-mail messages with EWS   
recipe earlier in this chapter. Add the following function to your shell session:

function Send-EWSMailMessage {

param(

[Parameter(

Position=0,

Mandatory=$true,

ValueFromPipelineByPropertyName=$true

)]

[String[]]

$PrimarySmtpAddress,

[Parameter(

Position=1, Mandatory=$true

)]

[String]

$From,

[Parameter(

Position=2, Mandatory=$true

)]

[String]

$Subject,

[Parameter(

Position=3, Mandatory=$true

)]

[String]

$Body,

[Parameter(

Position=4, Mandatory=$false

)]

[String[]]

$Cc,

[Parameter(

Position=5, Mandatory=$false

)]

[String[]]

$Bcc

)

begin {

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

}

process {

$svc = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$From

$svc.ImpersonatedUserId = $id

$svc.AutodiscoverUrl($From)

$msg = New-Object `

-TypeName Microsoft.Exchange.WebServices.Data.EmailMessage `

-ArgumentList $svc

$msg.Subject = $Subject

$msg.Body = $Body

$PrimarySmtpAddress | %{

$msg.ToRecipients.Add($\_) | Out-Null

}

if ($Cc -ne $null)

{

$msg.CcRecipients.Add($Cc) | Out-Null

}

if ($Bcc -ne $null)

{

$msg.BccRecipients.Add($Bcc) | Out-Null

}

$msg.SendAndSaveCopy()

}

}

As you can see, we've modified this version of the function so that the SMTP address specified using the -From parameter is used as the impersonated user ID. Let's say that you are logged into Windows using the domain administrator account, which has been assigned the ApplicationImpersonation RBAC role. Once the function has been loaded into the shell you could execute the following command:

Send-EWSMailMessage -From sysadmin@testlabs.se `

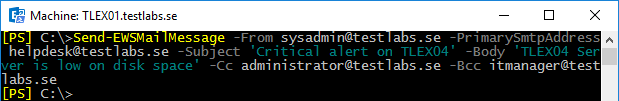
-PrimarySmtpAddress helpdesk@testlabs.se `

-Subject 'Critical alert on TLEX04' `

-Body 'TLEX04 Server is low on disk space' `  
-Cc administrator@testlabs.se `

-Bcc itmanager@testlabs.se

Using this command, the e-mail message is sent through EWS from the sysadmin mailbox. The message appears to the recipient as if the sysadmin account had sent it. The screenshot below shows an illustrative example on how it could be used:



7081EN\_14\_07

# Searching mailboxes

The EWS Managed API can be used to search one or more folders within an Exchange mailbox. The latest version of the API supports searches using Advanced Query Syntax, allowing us to search folders using the indexes created by the Exchange Search service.   
This makes searching a mailbox folder very fast and less resource intensive than methods that were used with first versions of the API. In this recipe, you'll learn how to search the contents of a mailbox through PowerShell and the EWS Managed API.

## How to do it...

1. First, load the assembly, create the ExchangeService object and connect to EWS:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("lsanders@testlabs.se")

2. Next, create a view for the total number of items that should be returned from the search:

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

3. The next step is to create a property set containing all the properties of each message we want returned, and then associate that property set with the $view object created in the last step:

$propertyset = New-Object Microsoft.Exchange.WebServices.Data.PropertySet (

[Microsoft.Exchange.WebServices.Data.BasePropertySet]::IdOnly,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::Subject,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::HasAttachments,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DisplayTo,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DisplayCc,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DateTimeSent,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DateTimeReceived

)

$view.PropertySet = $propertyset

4. Next, define a search query using AQS syntax:

$query = "Subject:'Critical alert on TLEX01'"

5. We can then perform the search using the FindItems method of our Exchange Service object:

$items = $svc.FindItems("Inbox",$query,$view)

6. Finally, loop through each item and return a custom object that contains the properties for each message:

$items | Foreach-Object{

New-Object PSObject -Property @{

Id = $\_.Id.ToString()

Subject = $\_.Subject

To = $\_.DisplayTo

Cc = $\_.DisplayCc

HasAttachments = [bool]$\_.HasAttachments

Sent = $\_.DateTimeSent

Received = $\_.DateTimeReceived

}

}

When executing this code, any of the last 100 items in the administrator inbox that have the word "sales" in the subject line will be returned.

## How it works...

Since we are not supplying credentials when creating the ExchangeService object and we're not using impersonation, the search will be performed in the administrator mailbox, as this is the logged-on user. You probably noticed that the property set only contains a few key properties of each message. Although there are many more available properties that can be returned, as a best practice we should only retrieve the properties that interest us. That way, if we are executing the code over and over, perhaps even against multiple mailboxes, we are not burdening the Exchange servers by requesting unnecessary data.

The key to a successful search is constructing the appropriate AQS query. You can use an AQS query for specific properties of a message using word phrase restriction, date range restriction, or message type restriction. For example, instead of querying using the Subject property, we can search for messages retrieved within a certain time frame:

$svc.FindItems(

"Inbox",

"Sent:05/01/2017..05/30/2017",

$view

)

Notice that the first value passed in the call to FindItems is the folder that we want to search, next is the AQS query that specifies that we only want to retrieve items that were   
sent between specific dates in May, and finally we pass in the $view object   
that specifies the total items to return with a defined property set.

There are a number of well-known mailbox folders that can be searched using the   
FindItems method:

* ArchiveDeletedItems: The Deleted Items folder in the archive mailbox
* ArchiveMsgFolderRoot: The root of the message folder hierarchy in the archive mailbox
* ArchiveRecoverableItemsDeletions: The root of the folder hierarchy of recoverable items that have been soft-deleted from the Deleted Items folder of the archive mailbox
* ArchiveRecoverableItemsPurges: The root of the hierarchy of   
  recoverable items that have been hard-deleted from the Deleted Items   
  folder of the archive mailbox
* ArchiveRecoverableItemsRoot: The root of the Recoverable Items folder hierarchy in the archive mailbox
* ArchiveRecoverableItemsVersions: The root of the Recoverable Items versions folder hierarchy in the archive mailbox
* ArchiveRoot: The root of the folder hierarchy in the archive mailbox
* Calendar: The Calendar folder
* Contacts: The Contacts folder
* DeletedItems: The Deleted Items folder
* Drafts: The Drafts folder
* Inbox: The Inbox folder
* JunkEmail: The Junk E-mail folder
* RecoverableItemsDeletions: The root of the folder hierarchy of recoverable items that have been soft-deleted from the Deleted Items folder
* RecoverableItemsPurges: The root of the folder hierarchy of recoverable items that have been hard-deleted from the Deleted Items folder
* RecoverableItemsRoot: The root of the Recoverable Items folder hierarchy
* RecoverableItemsVersions: The root of the Recoverable Items versions folder hierarchy in the archive mailbox
* SearchFolders: The Search Folders folder, also known as the Finder folder

SentItems: The Sent Items folder

For details, see the full list of members for the WellKnownFolderName enumeration in the Exchange Web Services Managed API SDK documentation on MSDN (even though this is for 2.0 it still applies to version 2.2):

http://msdn.microsoft.com/en-us/library/microsoft.exchange.webservices.data.wellknownfoldername(v=exchg.80).aspx

http://msdn.microsoft.com/en-us/library/jj536567(v=exchg.150).aspx

## There's more...

One piece of interesting information not returned by the code in the previous example is the body of the message. This is because there are a number of properties that the FindItems method will not return, one of which is the message body. In order to retrieve the message body, we can bind to the message after the search has been performed using the ID of the message.

Let's extend the previous code so that we can retrieve the body of the message and add the ability to impersonate the target mailbox. Add the following code to a file called MailboxSearch.ps1:

Param($query,$mailbox)

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$mailbox

$svc.ImpersonatedUserId = $id

$svc.AutoDiscoverUrl($mailbox)

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

$propertyset = New-Object Microsoft.Exchange.WebServices.Data.PropertySet (

[Microsoft.Exchange.WebServices.Data.BasePropertySet]::IdOnly,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::Subject,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::HasAttachments,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DisplayTo,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DisplayCc,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DateTimeSent,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::DateTimeReceived

)

$view.PropertySet = $propertyset

$items = $svc.FindItems("Inbox",$query,$view)

$items | Foreach-Object{

$emailProps = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet(

[Microsoft.Exchange.WebServices.Data.BasePropertySet]::IdOnly,

[Microsoft.Exchange.WebServices.Data.ItemSchema]::Body

)

$emailProps.RequestedBodyType = "Text"

$email = [Microsoft.Exchange.WebServices.Data.EmailMessage]::Bind(

$svc, $\_.Id, $emailProps

)

New-Object PSObject -Property @{

Id = $\_.Id.ToString()

Subject = $\_.Subject

To = $\_.DisplayTo

Cc = $\_.DisplayCc

HasAttachments = [bool]$\_.HasAttachments

Sent = $\_.DateTimeSent

Received = $\_.DateTimeReceived

Body = $email.Body

}

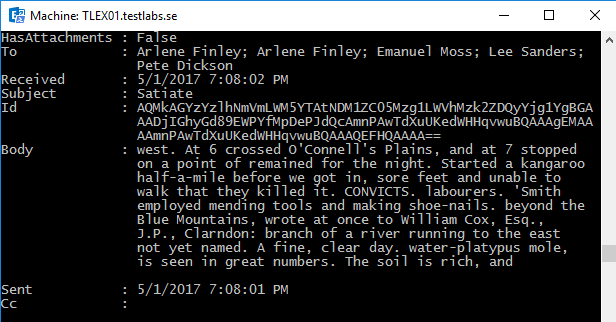
}

When running the script, provide values for the -Query and -Mailbox parameters:

c:\MailboxSearch.ps1 -query "Sent:05/01/2017..05/30/2017" `

-mailbox lsanders@testlabs.se

When the script executes, the first 100 items in the Lee Sanders mailbox that were sent between May 1st and 30th will be returned. The script will output a custom object for each item that contains the Id, Subject, To, Cc, HasAttachments, Sent, Received, and Body properties. Notice that, even though the body might be composed as HTML, we've only requested the text type for the body in the property set used when binding to the message. See screenshot below for similar example of the cmdlet above:



7081EN\_14\_08

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Notice that the date format is changed due to the regional settings in my lab environment, use the date format that applies to your environment. |  |

## See also

Exporting attachments from a mailbox

# Retrieving the headers of an e-mail message

When troubleshooting mail flow issues, you may need to take a look at the headers of an e-mail message. This is easy to do through Outlook for items in your own mailbox, but if you want to do this on behalf of another user, it requires you to have permissions to their mailbox, and then you need to open their mailbox in Outlook to view the headers. In this recipe, we'll take a look at how you can retrieve the headers of a message in your own mailbox, as well as another user's mailbox, using the EWS Managed API and PowerShell.

## How to do it...

1. First, load the assembly, create the ExchangeService object, and connect to EWS:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("lsanders@testlabs.se")

2. Next, create a view for the total number of items that should be returned from the search:

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

3. The next step is to create a property set that will include the message ID. We then need to associate that property set with the $view object created in the last step:

$schema = [Microsoft.Exchange.WebServices.Data.ItemSchema]

$propertyset = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$schema::IdOnly

)

$view.PropertySet = $propertyset

4. Next, define a search query using AQS syntax:

$query = "Subject:'Sales'"

5. We can then perform the search, using the FindItems method of our Exchange Service object:

$items = $svc.FindItems("Inbox",$query,$view)

6. Loop through each item returned by the search and retrieve the message header information:

$items | Foreach-Object{

$headerview = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 1

$headerprops = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$schema::InternetMessageHeaders

)

$headerview.PropertySet = $headerprops

$message = [Microsoft.Exchange.WebServices.Data.Item]::Bind(

$svc, $\_.Id, $headerview.PropertySet

)

$message.InternetMessageHeaders

}

## How it works...

The code in this example is very similar to what we used in the recipe for Searching mailboxes. Again, since we are not supplying credentials when creating the ExchangeService object, and we're not using impersonation, the search will be performed in the administrator mailbox. When calling the FindItems method, we're specifying the folder to search, the AQS search query to be used, and the item view.

For each item returned by the search, we need to create a new view and property set for the single instance of the message that returns only the message headers. We then bind to the message and return the header information.

The header information returned will provide details of which server received the message, the content type of the message, the subject and date, and all of the X-Headers included with the message.

There are a number of well-known mailbox folders that can be searched using the FindItems method. For details, see the recipe earlier in this chapter titled Searching mailboxes.

## There's more...

Of course, we'll primarily need to retrieve the message headers for an item in another user's mailbox. Here is an extended version of our previous code that implements EWS impersonation and provides parameters for the mailbox and folder to be searched. Add the following code to a script called GetMessageHeaders.ps1:

Param($query, $mailbox, $folder)

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$mailbox

$svc.ImpersonatedUserId = $id

$svc.AutoDiscoverUrl($mailbox)

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

$schema = [Microsoft.Exchange.WebServices.Data.ItemSchema]

$propertyset = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$schema::IdOnly

)

$view.PropertySet = $propertyset

$query = $query

$items = $svc.FindItems($folder,$query,$view)

$items | Foreach-Object{

$headerview = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 1

$headerprops = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$schema::InternetMessageHeaders

)

$headerview.PropertySet = $headerprops

$message = [Microsoft.Exchange.WebServices.Data.Item]::Bind(

$svc, $\_.Id, $headerview.PropertySet

)

$message.InternetMessageHeaders

}

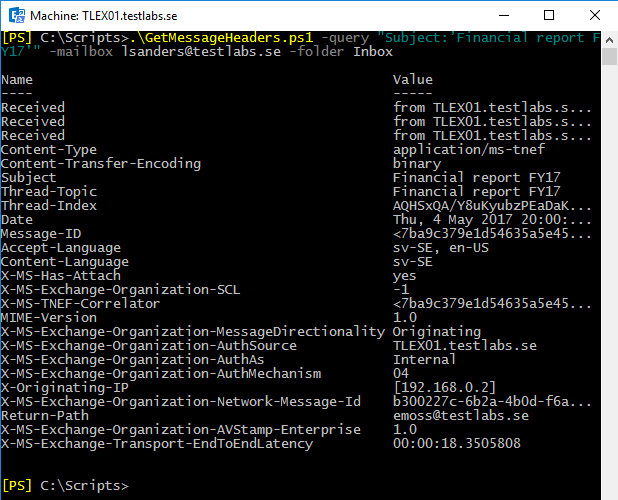
To run the script against an alternate mailbox, provide the query and the SMTP address associated with the mailbox:

c:\GetMessageHeaders.ps1 -query "Subject:’Financial report FY17’"`

-mailbox lsanders@testlabs.se `

-folder Inbox

When the script executes, the headers for each message matching the AQS query will   
be returned and the output will look similar to the screenshot below:



7081EN\_14\_09

## See also

Working with impersonation

# Deleting e-mail items from a mailbox

The Exchange Management Shell provides cmdlets that allow you to delete items from one or more mailboxes. This can also be done with the EWS Managed API, and you can get a little more control over how the items are deleted compared to what the built-in cmdlets provide. In this recipe, you'll learn how to use the EWS Managed API to delete items from one or more mailboxes using PowerShell.

## How to do it...

1. First, load the assembly, create the ExchangeService object, and connect to EWS:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("administrator@testlabs.se")

2. Next, create a view for the total number of items that should be returned from the search:

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

3. Create a property set that will include the message id. We then need to associate that property set with the $view object created in the last step:

$propertyset = New-Object Microsoft.Exchange.WebServices.Data.PropertySet (

[Microsoft.Exchange.WebServices.Data.BasePropertySet]::IdOnly

)

$view.PropertySet = $propertyset

4. Next, define a search query using AQS syntax:

$query = "body:'Inappropriate content'"

5. We can then perform the search using the FindItems method of our Exchange Service object:

$items = $svc.FindItems("Inbox",$query,$view)

6. For each item returned by the search, bind to the message and call the Delete method, specifying the delete mode that should be used:

$items | Foreach-Object{

$message = [Microsoft.Exchange.WebServices.Data.Item]::Bind(

$svc, $\_.Id

)

$message.Delete("SoftDelete")

}

## How it works...

The code in this example is very similar to what we used in the recipe for Searching mailboxes. Again, since we are not supplying credentials when creating the ExchangeService object and we're not using impersonation, the search will be performed in the administrator mailbox. When calling the FindItems method, we're specifying the folder to search, the AQS search query to be used, and the item view.

Notice that this time we only need to specify the ID of the message in the property set. This is because we only want to call the Delete method on the item class and we don't need to retrieve any other data from the message. In this example, we've defined a string of inappropriate content that should be found in the message body.

We loop through each item returned by the search and create an instance of the message using the item class Bind method. At that point, we call the Delete method, which accepts one of three values from the DeleteMode enumeration. The valid values for this method are defined as follows:

* HardDelete: Permanently deletes the item
* MoveToDeletedItems: Moves the item to the Deleted Items folder of the target mailbox

SoftDelete: The item is moved to the dumpster and can be recovered by the mailbox owner using the Recoverable Items feature of Outlook and OWA

Having the ability to specify the delete mode gives you a little more control when deleting items in a mailbox than the built-in Exchange Management Shell cmdlets.

There are a number of well-known mailbox folders that can be searched for using   
the FindItems method. For details, see the recipe earlier in this chapter titled   
Searching mailboxes.

## There's more...

Whenever you are executing code that can perform a destructive operation, it makes sense to implement the ShouldProcess method introduced with PowerShell v2 advanced functions. Implementing ShouldProcess in an advanced function gives you the ability to add the common risk mitigation parameters such as -Whatif and   
-Confirm. The following function takes our previous code up a notch, written as an advanced function that implements ShouldProcess. Add the following function to your Exchange Management Shell session:

function Remove-MailboxItem {

[CmdletBinding(

SupportsShouldProcess = $true, ConfirmImpact = "High"

)]

param(

[Parameter(

Position=0,

Mandatory=$true,

ValueFromPipelineByPropertyName=$true

)]

[String]

$PrimarySmtpAddress,

[Parameter(

Position = 1, Mandatory = $true

)]

[String]

$SearchQuery,

[Parameter(

Position = 2, Mandatory = $false

)]

[int]

$ResultSize = 100,

[Parameter(

Position = 3, Mandatory = $false

)]

[string]

$Folder = "Inbox",

[Parameter(

Position = 4, Mandatory = $false

)]

[ValidateSet(

'HardDelete',

'SoftDelete',

'MoveToDeletedItems'

)]

$DeleteMode = "MoveToDeletedItems"

)

begin {

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

}

process {

$svc = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$PrimarySmtpAddress

$svc.ImpersonatedUserId = $id

$svc.AutoDiscoverUrl($PrimarySmtpAddress)

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

$propertyset = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

[Microsoft.Exchange.WebServices.Data.BasePropertySet]::IdOnly

)

$view.PropertySet = $propertyset

$items = $svc.FindItems($Folder,$SearchQuery,$view)

$items | %{

$message = [Microsoft.Exchange.WebServices.Data.Item]::Bind(

$svc, $\_.Id

)

if ($pscmdlet.ShouldProcess($message.Subject)) {

$message.Delete($DeleteMode)

}

}

}

}

We now have a Remove-MailboxItem function that supports impersonation, allowing the code to execute against one or more mailboxes. In addition, it supports pipeline input by property name, so you can utilize the Get-Mailbox cmdlet to delete items from multiple mailboxes using a simple one-liner. Consider the following example:

Get-Mailbox -ResultSize Unlimited |

Remove-MailboxItem -SearchQuery "Body:’Free Surface Book’" `

-DeleteMode HardDelete

In this example, we pipe every mailbox in the organization down to the   
Remove-MailboxItem function , which will perform a hard delete on each message that matches the AQS query. Since the ConfirmImpact property is set to High, you'll be prompted for confirmation before each message is deleted.

To force a delete operation without confirmation, you can set the –Confirm parameter to $false. To do this on a single mailbox, you could use the following syntax:

Remove-MailboxItem -PrimarySmtpAddress sysadmin@contoso.com `

-SearchQuery “Body:’Buy cheap drugs’” `

-DeleteMode HardDelete `

-Confirm:$false

You can also use the -Whatif parameter here to test the command to ensure that the correct messages will be deleted:

Remove-MailboxItem -PrimarySmtpAddress sysadmin@contoso.com `

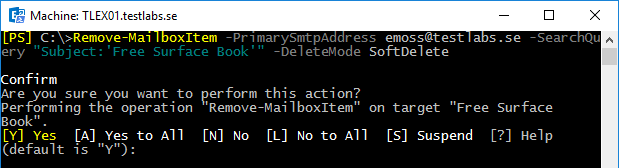
-SearchQuery “Body:’Buy cheap drugs’” `

-DeleteMode HardDelete `

-Whatif

To illustrate an example, we use the cmdlet Get-Mailbox and pipe the result into the created function together with a search query that searches for all mails with a subject that has ‘Free Surface Book’ included. The function will softdelete the mails that are returned, which means they will end up in the dumpster.

See the screenshot below for a similar output using the created function:



7081EN\_14\_10

## See also

Searching mailboxes

# Creating calendar items

Imagine that you have a monitoring script written in PowerShell that checks memory, CPU, or disk utilization on all of your Exchange servers. In addition to alerting your team of any critical problems via e-mail, it might also be nice to schedule a reminder in the future for non-critical issues by creating a calendar item in one or more mailboxes. The EWS Managed API makes it easy to create a calendar item through PowerShell with just a few commands.

## How to do it...

1. First, load the assembly, create the ExchangeService object, and connect to EWS:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("administrator@testlabs.se")

2. Next, create a new Appointment object:

$appt = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.Appointment `

-ArgumentList $svc

3. Fill out the subject and body for the appointment:

$appt.Subject = "Review Disk Space Utilization on Server(s)"

$appt.Body = "TLEX01 has only 40% free disk space on drive C:"

4. Set the start and end times for the appointment:

$start = (Get-Date).AddDays(1)

$appt.Start = $start

$appt.End = $start.AddHours(1)

5. Add one or more required attendees to the appointment:

$appt.RequiredAttendees.Add("helpdesk@testlabs.se")

$appt.RequiredAttendees.Add("sysadmin@testlabs.se")

6. Finally, save the appointment and send a copy to all attendees:

$mode = [Microsoft.Exchange.WebServices.Data.SendInvitationsMode]

$appt.Save($mode::SendToAllAndSaveCopy)

## How it works...

Using the code in this example, we are creating the calendar item in the mailbox of the   
user calling the code. The Appointment class is used to create the item and, after we've created an instance of this class, we set the details of the appointment using the Subject and Body properties.

The Start and End properties need to be assigned a DateTime object. In our example, we're using the AddDays method of the current date and time to set the start time for the meeting in exactly 24 hours in the future. We then use the same object to increment the   
time by one hour and assign that to the End property for the appointment.

When adding attendees to the appointment, we use the RequiredAttendees.Add method. When you call the Add method, you'll notice that the RequiredAttendees property will be returned for each required attendee added to the appointment. If you want to simply call this method without having anything returned to the screen, there's a few ways you can accomplish this. First, you can pipe the command to Out-Null:

$appt.RequiredAttendees.Add("helpdesk@testlabs.se") | Out-Null

$appt.RequiredAttendees.Add("sysadmin@testlabs.se") | Out-Null

Another way you'll see this written is by casting the commands to [void]:

[void]$appt.RequiredAttendees.Add("helpdesk@testlabs.se")

[void]$appt.RequiredAttendees.Add("sysadmin@testlabs.se")

Finally, you can assign the commands to $null, which is said to be the fastest method:

$null = $appt.RequiredAttendees.Add("helpdesk@testlabs.se")

$null = $appt.RequiredAttendees.Add("sysadmin@testlabs.se")

In addition to adding required attendees, we can also add one or more optional attendees   
to the item:

$null = $appt.OptionalAttendees.Add("IT@testlabs.se")

Finally, when calling the Save method for the appointment, you need to pass in a value from the SendInvitationsMode enumeration. The valid values that can be used are SendOnlyToAll, SendToAllAndSaveCopy, and SendToNone.

## There's more...

Let's make this easier by wrapping all of the code up into a reusable function. Add the following code to your PowerShell session:

function New-CalendarItem {

[CmdletBinding()]

param(

[Parameter(

Position=1, Mandatory=$true

)]

[String]

$Subject,

[Parameter(

Position=2, Mandatory=$true

)]

[String]

$Body,

[Parameter(

Position=3, Mandatory=$true

)]

[String]

$Start,

[Parameter(

Position=4, Mandatory=$true

)]

[String]

$End,

[Parameter(

Position=5

)]

[String[]]

$RequiredAttendees,

[Parameter(

Position=8

)]

[String]

$Mailbox

)

begin{

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

}

process {

$svc = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$Mailbox

$svc.ImpersonatedUserId = $id

$svc.AutodiscoverUrl($Mailbox)

$appt = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.Appointment `

-ArgumentList $svc

$appt.Subject = $Subject

$appt.Body = $Body

$appt.Start = $Start

$appt.End = $End

if($RequiredAttendees) {

$RequiredAttendees | Foreach-Object{

$null = $appt.RequiredAttendees.Add($\_)

}

}

$mode = [Microsoft.Exchange.WebServices.Data.SendInvitationsMode]

$appt.Save($mode::SendToAllAndSaveCopy)

}

}

This function can be used to create a calendar item in the mailbox of another user. For this to work, you'll need to be assigned the ApplicationImpersonation RBAC role. To run the function, you might do something like this:

New-CalendarItem -Subject "Reboot Server" `

-Body "Reboot EXCH-SRV01 server after 5PM today" `

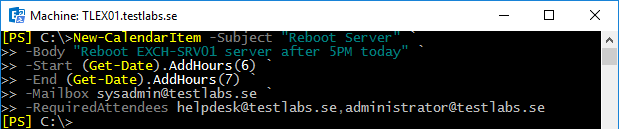
-Start (Get-Date).AddHours(6) `

-End (Get-Date).AddHours(7) `

-Mailbox sysadmin@testlabs.se `

-RequiredAttendees helpdesk@testlabs.se,admins@testlabs.se

In this example, the calendar item is created in the sysadmin mailbox. Multiple attendees will be added to the item and will receive an invitation for the meeting when it is saved. Notice that the meeting is scheduled for six hours in the future, with a total duration of one hour. For example, see the screenshot below:



7081EN\_14\_11

If you want to create calendar items in multiple mailboxes, loop through a collection with the Foreach-Object cmdlet and run the function for each user:

$start = Get-Date "Friday, May 26, 2017 5:00:00 PM"

$end = $start.AddHours(2)

Get-DistributionGroupMember ITSupport | Foreach-Object{

New-CalendarItem -Subject "Install Hotfixes" `

-Body "Start patching servers after 5PM today" `

-Start $start `

-End $end `

-Mailbox $\_.PrimarySMTPAddress

}

In this example, each member of the IT Support distribution group will have a calendar item created in their mailbox that will serve as a reminder; no attendees will be added to the item.

## See also

Sending e-mail messages with EWS

# Exporting attachments from a mailbox

The Exchange Management Shell provides cmdlets that allow you to export e-mail messages from one mailbox to another mailbox. These e-mails can then be exported to a PST file, or you can open an alternate mailbox and access the data. The only limitation is that this provides no option to export only the message attachments. The EWS Managed API has this functionality built in. In this recipe, you'll learn how to export e-mail attachments from an Exchange mailbox using PowerShell.

## How to do it...

1. First, load the assembly, create the ExchangeService object, and connect to EWS:

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$svc.AutoDiscoverUrl("administrator@testlabs.se")

2. Next, create a view for the total number of items that should be returned from the search:

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

3. Next, create a property set and then associate that property set with the $view object:

$base = [Microsoft.Exchange.WebServices.Data.BasePropertySet]

$propertyset = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$base::FirstClassProperties

)

$view.PropertySet = $propertyset

4. Define a query for the type of attachments you are looking for. For example, if you are looking for attachments in Microsoft Word format, use the following:

$query = "Attachment:docx"

5. We can then perform the search using the FindItems method of our Exchange Service object:

$items = $svc.FindItems("Inbox",$query,$view)

6. Finally, we loop through each item returned and export the attachments to the specified folder on the file system, such as c:\export:

$items | ForEach-Object{

if($\_.HasAttachments ) {

$\_.Load()

$\_.Attachments | ForEach-Object {

$\_.Load()

$filename = $\_.Name

Set-Content -Path c:\export\$filename `

-Value $\_.Content `

-Encoding Byte `

-Force

}

}

}

## How it works...

The code in this example is very similar to what we used in the recipe for Searching Mailboxes. Again, since we are not supplying credentials when creating the ExchangeService object and we're not using impersonation, the search will be performed in the administrator mailbox. When calling the FindItems method, we're specifying the folder to search, the AQS search query to be used, and the item view.

As you can see, we're using the Attachment property in the AQS query. This allows us to search for a string within the file name or inside the file itself. When the results are returned, we loop through each message, and use the Load method to load the attachment, which allows us to then access the Content property of each attachment. The Content property stores the message attachment as a byte array, which can easily be used to recreate the file using the Set-Content cmdlet by specifying the encoding as Byte.

There are a number of well-known mailbox folders that can be searched using the FindItems method. For details, see the recipe earlier in this chapter titled Searching mailboxes.

## There's more...

Like many of our previous examples, reusability is key. Let's take this code and add a few enhancements so it can be run via a PowerShell script. Add the following code to a file called AttachmentExport.ps1:

Param($folder, $query, $path, $mailbox)

Add-Type -Path C:\EWS\Microsoft.Exchange.WebServices.dll

$svc = New-Object Microsoft.Exchange.WebServices.Data.ExchangeService

$id = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ImpersonatedUserId `

-ArgumentList "SmtpAddress",$mailbox

$svc.ImpersonatedUserId = $id

$svc.AutoDiscoverUrl($mailbox)

$view = New-Object -TypeName `

Microsoft.Exchange.WebServices.Data.ItemView `

-ArgumentList 100

$base = [Microsoft.Exchange.WebServices.Data.BasePropertySet]

$propertyset = New-Object –TypeName `

Microsoft.Exchange.WebServices.Data.PropertySet (

$base::FirstClassProperties

)

$view.PropertySet = $propertyset

$items = $svc.FindItems($folder,$query,$view)

$items | Foreach-Object{

if($\_.HasAttachments) {

$\_.Load()

$\_.Attachments | ForEach-Object {

$\_.Load()

$filename = $\_.Name

Set-Content -Path $path\$filename `

-Value $\_.Content `

-Encoding Byte `

-Force

}

}

}

Using this script, we can export the attachments from one or more mailboxes since we've included the code to support impersonation. Just make sure your account has been assigned the ApplicationImpersonation RBAC role when running this script against another mailbox. Let's say we wanted to export all of the Excel files that are attached to messages in the sysadmin mailbox. Run this script with the following syntax:

c:\AttachmentExport.ps1 -folder Inbox `

-mailbox emoss@testlabs.se `

-query "attachment:docx" `

-path c:\Export

You can also export all attachments simply by using a wildcard in the search query:

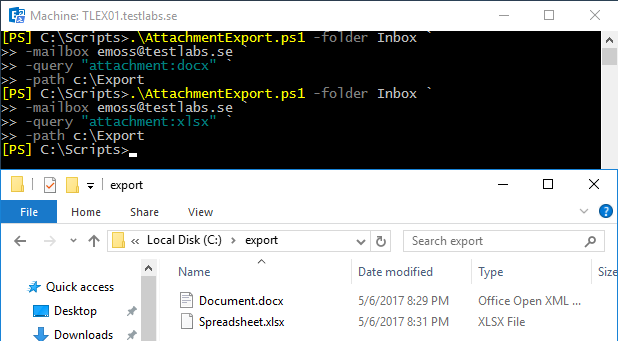
c:\AttachmentExport.ps1 -folder Inbox `

-mailbox emoss@testlabs.se `

-query "attachment:xlsx" `

-path c:\Export

Keep in mind that, since our item view is set to 100, we may need to increase the number if we want to search through mailbox folders with a higher item count. See screenshot below for example usage of the script and the result:



7081EN\_14\_12

## See also

Searching mailboxes