PowerShell – Copy Exchange 2010 Receive Connectors Between Servers

The situation that many Exchange administrators are in, is a simple one. Multiple CAS servers for redundancy and fault tolerance, as well as load balancing either inside the cluster, or on the outside using something like a Citrix Netscaler or F5 device. This is a natural design for Exchange 2010 and provides great functionality and recovery capability.

There are also many designs which use a single server, or perhaps a single server per site contains Receive Connectors for inbound relay based on a set of criteria such as auth type or network range.

The challenge for Receive Connectors, and especially with network ranges applied, is that they are difficult to re-create on another server. Whether it is on your first creation, or if you want to make a copy on another server for active or passive use, PowerShell can be your best friend for this task.

Assuming that we have two servers named **EXSITE1** and **EXSITE2** where we have created our Receive Connector named ***Default-App-Connector*** on **EXSITE1** but we want to have it on **EXSITE2**. While we can create one easily enough using the **New-ReceiveConnector** CmdLet we have to type in all of the IP ranges manually which is both tedious and error prone.

Here is your solution. The basic command for creating the new backup connector is:

```
New-ReceiveConnector “Default-App-Connector” -Server EXSITE2 -Bindings 0.0.0.0:25
```

The problem is that this only creates the connector, but not the IP ranges. As I mentioned, if we type the allowed IP addresses and ranges into the command using the ***-RemoteIPRanges*** parameter I have a lot of work ahead of me.

So we simply read the ***-RemoteIPRanges*** from the first connector and pass them to the **New-ReceiveConnector** CmdLet just like so:

```
New-ReceiveConnector “Default-App-Connector” -Server EXSITE2 -Bindings 0.0.0.0:25 -
RemoteIPRanges ( Get-ReceiveConnector “EXSITE1Default-App-Connector”
).RemoteIPRanges
```

You can also use the **Get-ReceiveConnector** to document your configuration to file, which is a good practice for BCP. Because these can be volatile, I recommend you export to disk, or replicate to the second server weekly or monthly.

Simply use this command:

```
Get-ReceiveConnector “EXSITE1Default-App-Connector” | Format-List | Out-File
“X:ExchangeConfigurationDefault-App-Connector.txt”
```

It’s just that easy. A simple command that can provide peace of mind and protection. Hopefully you

find this to be helpful.

Exchange 2010 SP2 and Citrix Netscalers – SSL gotcha

This is a quick note about a real issue that I ran into very recently. With the Microsoft Exchange 2010 SP2 update there are a number of things to be careful of. With any of the major rollups and service packs, there are often default configuration settings which are re-enabled as a part of the update.

If you've been running Exchange with Outlook Web Access then you have most likely hit the first of two issues which is that any customized OWA pages (logon, logoff etc.) are overwritten during updates. Make sure that you save any custom configurations, and if you have configured a different default theme to be used by your CAS servers in 2010, you may also have to reset the default again to your chosen theme.

The second issue which will really twist you up is if you are using a Network Load Balancer (NLB) such as the Citrix Netscaler appliance. The reason that this is an issue is because of the SSL offload capability which is one of the great advantages of these devices.

During the update of the CAS roles to SP2, the Default Web Site suddenly disappeared from the active monitoring despite the fact that the site itself was still up when accessed directly from the local server using the https://yourservername/owa URL. The key to this is that during the update the SSL option was re-enabled on the Default Web Site for the CAS server.

Because we use the NLB for managing the SSL we can safely uncheck the SSL Required option within IIS and as if by magic, the site is now available again through the load balanced configuration.

Are you sure? Using the -WhatIf and -Confirm parameters in PowerShell

There are 2 very helpful parameters you can apply to many PowerShell CmdLets. These are the –**WhatIf** and the –**Confirm** options. There are many cases where you are processing a number of

records which you would like to get an idea on how to “test drive” the process on a large chunk of data.

First, by utilizing the –**WhatIf** parameter you can preview the changes that will take place against the object without actually committing those changes. This is a great way to ensure that your process or script will give you the expected results.

The second very handy parameter is –**Confirm** which will pass information to the script where it prompts for confirmation before proceeding with the command. You will find that many, if not most, delete commands have this option attached so that a runaway delete process doesn’t just blindly wipe out data.

The –**WhatIf** switch is self-explanatory. Here is my data, so What If I run a command against it? In the following script, I’m pushing some variables into a **Set-Contact** CmdLet for Exchange 2010. I want to ensure that the updates will go without issue, so I tag the command with the –**WhatIf** option and the result will show me that the command with either pass and perform updates, or if there is an issue, it will throw an error.

> **Set-Contact -Identity $sourceEmail -City $sourceAddressCity -Company $sourceBranchLegalName -CountryOrRegion $sourceAddressCountry -WhatIf**

Now we want to see a delete process, which in PowerShell are almost always “Remove” CmdLets. On a side note, I would definitely prefer that PowerShell followed the CRUD or Create Read Update Delete convention which we see referenced in application and web development. But I digress, so let’s get back to the task at hand.

I want to remove a contact record which matches a variable which I’ve named $sourceEmail that I’m reading from somewhere. The TechNet help (http://technet.microsoft.com/en-us/library/bb123561.aspx) on the **Remove-MailContact** CmdLet shows the following description for the –**Confirm** parameter:

> The *Confirm* switch causes the command to pause processing and requires you to acknowledge what the command will do before processing continues. You don’t have to specify a value with the *Confirm* switch.

So if we read this correctly, the default behaviour of the CmdLet is to *not* prompt for confirmation, but if you want it to prompt you before taking action you need to add the –**Confirm** to your command. So we run the command as-is so that it runs without interaction like this…right?:

> **Remove-MailContact -Identity “$sourceEmail”**

But here is the catch! The command still prompts you despite the fact that you have not explicitly required confirmation using the very information that Microsoft themselves have provided. As in the magic world, this is the Turn where we specifically ask the CmdLet to prompt, but we pass data to it within the command:

> **Remove-MailContact -Identity “$sourceEmail” -Confirm:$Y -WhatIf**

The result of this command is that we explicitly prompt for the confirmation, and the **$Y** sends a **Y** character to the prompt, thus answering the nagging question at the console for us.

I hope that you find these two little parameters to be friendly and helpful. Most importantly, make sure that you always test your processes in a non-production environment first for the best possible level of safety before you put them into production.

If only everything in life came with a -WhatIf option.