# NetIQ Directory and Resource Administrator - PowerShell Usage and Examples Reference

August 2020

This paper highlights how to use PowerShell to write DRA Triggers, DRA Custom Policies, standalone scripts that use the DRA ADSI Provider, and scripts that issue requests directly to DRA servers. Detailed information regarding the use of PowerShell can be found at the Microsoft Developer Network web site. This paper does not discuss the REST features allowing access to DRA servers.



## Binding to an Object Using the DRA ADSI Provider in a PowerShell Script

When you run a DRA server on a 64-bit Windows platform, you must use the version of PowerShell located in the **\Windows\SysWOW64** folder.

To bind to the Users generic container object in the NQTraining domain, use the following PowerShell statement.

```
$objContainer = [ADSI]"OnePoint://netiqwin2k8r20/CN=Users,DC=nqtraining,DC=lab"
```

**NOTE:** Specifying `netiqwin2k8r20` identifies `netiqwin2k8r20` as the DRA server to which the request will be directed. If a DRA server is omitted along with the training "/", the ADSI provider will choose a DRA server from among the available DRA servers.

# Checking for Errors in a PowerShell Script

By using the trap construct, you can implement behavior in DRA Triggers and Custom Policies corresponding to the "On Error Resume Next" mechanism offered by the VBScript engine. Specifically, by including the following at the beginning of your PowerShell scripts, terminating and non-terminating errors can be ignored but logged.

```
$ErrorActionPreference = "SilentlyContinue"

$Error.Clear()

trap { continue }
```

**NOTE:** Depending on the status of the PowerShell environment on a particular DRA server, you may not need to assign a value to $ActionPreference.

Error is a PowerShell object you do not need to declare. It functions to record errors that occur as a PowerShell Trigger or Custom Policy executes. Error can be accessed in much the same way as an array. For example, `$Error[0]`.

**NOTE:** The trap construct may not be able to recognize all errors. In particular, executing a statement such as `$v = 1/0` will result in an unrecoverable error.

# Creating an Object

The following fragment shows how a new user object can be created using the DRA ADSI Provider:

```
# netiqwin2k8r20 below identifies a DRA server.  If a server name is omitted, the provider will choose # a DRA server from among the servers in the multi-master set supporting the domain

$objContainer =  [ADSI]"OnePoint://netiqwin2k8r20/cn=Users,DC=nqtraining,DC=lab"

$objUser = $objContainer.Create("user", "cn=Jack Jones")

$objUser.Put("userPrincipalName", "jjones@central.com")

$objUser.Put("sAMAccountName", "jjones")

# Additional attributes and their values can also be specified using the Put method.

# Note that when specifying values for passwords, you must use the PutEncrypted method.

$password = 'P@ssw0rd'

$objUser.PutEncrypted("userPassword",  "P@ssw0rd") # currently not functioning

objUser.SetInfo()
```

# Deleting an Object

The statements below fail, even though the three statements succeed if OnePoint is changed to LDAP, after removing the DRA server name.

```
$objContainer = [ADSI]   "OnePoint://netiqwin2k8r20/cn=Users,DC=nqtraining,DC=lab"

$objContainer.Delete("user", "cn=user1x")  # currently not functioning-functions
w/LDAP provider

$objContainer.Delete("contact", "cn=cntct1")  # currently not functioning-
functions w/LDAP provider
```

# Determining the Properties of an Object

The statements below retrieve and display the sAMAccountName and userPrincipalName for a user account.

```
$objU1x = [ADSI]"OnePoint://netiqwin2k8r20/
cn=user1x,CN=Users,DC=nqtraining,DC=lab"

$sam = $objU1x.Get('sAMAccountName')

$sam

$up   = $objU1x.Get('userPrincipalName')

$up
```

# Enumerating Objects

Object enumeration involving ADSI filters seems not to function correctly. Please see the DRA SDK for examples.

# Getting Object Properties with the GetInfoEx Method

Please see the DRA SDK for examples that can be rewritten as PowerShell scripts.

# Setting Object Properties

The following is an example of a fragment that modifies the value of the initials attribute of a user account.

```
$objU1x = [ADSI]"OnePoint://netiqwin2k8r2/cn=Bob
Slydell,CN=Users,DC=nqtraining,DC=lab"

$initials = $objU1x.Get('initials')

$initials = $initials.ToUpper();

$objU1x.Put('initials', $initials)

$objU1x.SetInfo()
```

# Working with Resource Objects

Please see the DRA SDK for examples that can be rewritten as PowerShell scripts.

# Writing DRA Triggers and Custom Policies as PowerShell Scripts

DRA 8.7 supports Triggers and Custom Policies as PowerShell scripts. These scripts execute on DRA servers using the PowerShell engine installed on those servers. PowerShell Triggers and Custom Policies succeed or fail depending on a Boolean value that is returned. For example:

```
return $true # returns control to the DRA server signaling success

return $false # returns control to the DRA server signaling failure
```

To prevent the execution of malicious scripts, PowerShell enforces an execution policy. By default, the execution policy is set to Restricted, which means that PowerShell scripts will not run. You can determine the current execution policy by using the following cmdlet:

```
Get-ExecutionPolicy
```

The execution policies you can use are:

**Restricted:** Scripts will not run.

**RemoteSigned:** Scripts created locally will run, but those downloaded from the Internet will not run unless they are digitally signed by a trusted publisher.

**AllSigned:** Scripts will only run if they have been signed by a trusted publisher.

**Unrestricted:** Scripts will run regardless of their origin and whether they are signed.

---

**NOTE:** You can set PowerShell's execution policy by using the following cmdlet:

```
Set-ExecutionPolicy <policy name>
```

---

The examples and fragments described in this paper were executed on a DRA server after the following PowerShell cmdlet had been executed at a PowerShell command prompt as an administrator of the DRA server:

```
Set-ExecutionPolicy Unrestricted
```

When PowerShell DRA Triggers and Custom Policies execute, InVarSet requires no declaration and is initialized to the contents of the VarSet object.

Varset exposes following methods:

Object, `InVarSet.Get(<string key>)`: Retrieves a `value` from InVarSet. Null is returned if the key does not exist in the varset.

Void `InVarSet.Put(string key, value)`: Adds or updates a `value` in inVarSet. If key already exists in InVarSet, its value will be updated, if not it will be added.

Void `InVarSet.Put(string key, string[] value)`: Adds or updates a `string[]` in InVarSet. If the key already, its value will be updated, if not it will be added.

Void `InVarSet.Put(string key, object[] value)`: Adds or updates an `object[]` in InVarSet. If key already exists, its value will be updated, if not it will be added.

Void `InVarSet.PutEncrypted(string key, object value)`: Adds or updates an encrypted value in the VarSet. If key already exists, its value will be updated, if not it will be added.

Void `InVarSet.Remove(string key)` ::: Removes a key and all subkeys from InVarSet.

Void `InVarSet.Clear()`: Removes all keys and values from InVarSet. In practice, this method will rarely be used in a Trigger or Custom Policy.

Void `InVarSet.DumpToFile(string filename)`: Writes InVarSet data to a human-readable log file.

The statements below could be collected into a file having a `.ps1` extension and installed as a DRA Pre-Task Trigger for the operation UserCreate. This text is just intended to illustrate some of the features of PowerShell Triggers and Custom Policies and does not represent any sort of recommendation.

```
# Error recovery in PowerShell Triggers and Custom Policies can be handled using

# the PowerShell "try/catch/finally" mechanism.  In addition, using the next three lines can

# offer behavior similar to the error recovery mechanism currently supporting VBScript triggers.


$ErrorActionPreference = "SilentlyContinue"

$Error.Clear()

trap { continue }

# Creating a File where text can be directed.  (Although you can use this approach to collect debugging # data, conflicts can arise if multiple instances of a trigger execute at the same time.)

Set-Content -Value "DRAPretask" -Path C:\DRAPretask.txt


# Creating an event source for a Windows log.  (Directing debugging text to a Windows log, avoids

# potential conflicts since the operating system manages log output even if multiple instances

# of a trigger execute at the same time.)

New-EventLog -LogName Application -Source DRATriggers

Write-EventLog -LogName Application -Source DRATriggers -EventId 1001 -Message "From PretaskA"

$zero = 0

$v = 1/$zero

#$Error.Count

#$Error[0]

Add-Content -Value $Error.Count -Path C:\DRAPretask.txt

Add-Content -Value $Error[0] -Path C:\DRAPretask.txt

$gsScriptName = "PreTaskA.ps1";

$gsErrorMsgFirstLine = "Automation trigger script: " + $gsScriptName;
```

```
$FirstArg = ""

$initials = ""

# $InVarset.Put("Out.ErrorMsg.Script", $gsErrorMsgFirstLine + " exiting at the
beginning!!!");

# return $false

#For retrieving the commandline agruments

$CmdLineArgs = $InVarSet.Get("CmdLine")

Add-Content -Value $CmdLineArgs -Path C:\DRAPretask.txt

#To retrieve the number of arguments in the argument string

$number = $InVarSet.Get("CmdLine.numArgs")

#Adding the content into file

Add-Content -Value $number -Path C:\DRAPretask.txt

$InVarSet.DumpToFile("C:\vsdump.txt")

# return $True

if($number -ne 0) {

    #To retrieve an individual argument and also if required to retrieve multiple
arguments


    $FirstArg = $InVarSet.Get("CmdLine.arg0")

    $initials = $InVarSet.Get("CmdLine.arg1")

    Add-Content -Value $FirstArg -Path C:\DRAPretask.txt

    $Message = "From PretaskA: FirstArg: " + $FirstArg

   Write-EventLog -LogName Application -Source DRATriggers -EventId 1001 -Message
$Message

}

# $InVarset.Put("Out.ErrorMsg.Script", $gsErrorMsgFirstLine + " before return
false!");

# return $False

if($FirstArg -ne "Arg0x")  {

    $InVarset.Put("Out.ErrorMsg.Script", $gsErrorMsgFirstLine + " before Arg0x
check!");

    return $FALSE

}


   $sOperationName = $InVarset.Get("In.OperationName");

   if($sOperationName.length -eq 0)   {
```

```
      $sErrorMsgText =  " OperationName not retrieved from $InVarset.";
      $InVarset.Put("Out.ErrorMsg.Script", $gsErrorMsgFirstLine + $sErrorMsgText);
      return $False;
}
   # Updating the description property value based on arguments
   $InVarset.put("In.Properties.description", $FirstArg)
   Add-Content -Value " Here 1" -Path C:\DRAPretask.txt
   #Updating the initials property value
   $InVarset.put("In.Properties.initials", $initials)
   Add-Content -Value " Here 2" -Path C:\DRAPretask.txt
  $InVarset.Put("Out.WarningMsg.Script", $gsErrorMsgFirstLine + " Warning message
text here!!!");
  $Message = "From PretaskA: Warning: " + $gsErrorMsgFirstLine + " Warning message
text here!!!"
   Write-EventLog -LogName Application -Source DRATriggers -EventId 1001 -Message
$Message
   # return $True
   $sNewName = "";
   if( $InVarset.Get("In.Properties.sn"))
     {
          if( $InVarset.Get("In.Properties.givenName")) {
             $sNewName = $InVarset.Get("In.Properties.sn").Trim() + ", " +
             $InVarset.Get ("In.Properties.givenName").Trim()
             $InVarset.put("In.Properties.cn", $sNewName)
       }
       else
      {
         $sNewName = $InVarset.Get("In.Properties.sn").Trim();
      }
      $InVarset.Put("Out.WarningMsg.Script", $gsErrorMsgFirstLine + " Warning
message text!!!");
      Add-Content -Value " Before dump" -Path C:\DRAPretask.txt
      $InVarSet.DumpToFile("C:\vsdump.txt")
      return $True;
      }
      else
```

```
{
return $True;
}
```

## Issuing Request through PowerShell Using DRA COM Objects

The example script below instantiates several DRA COM objects installed on DRA servers and DRA client computers and uses them to retrieve several attributes of a user object.

```
# Get an instance of a DRA Connector
$sServer = "netiqwin2k8r20"
$gWebDcom = New-Object -ComObject "McsWebDcom.Connector.1"
# Get an instance of an EAServe object.  $sServer in the next
# statement identifies a computer running the NetIQ Administration Service
$gEaServerObject = $gWebDcom.GetEaServer($sServer)
# Create an instance of the VarSet object to be used to contain inputs to a DRA request
$VarSetIn = New-Object -ComObject "NetIQDRAVarSet.VarSet.1"
# Issue the Put method on VarSetIn.  This method accepts two parameters.
# The first is a string that specifies the name for the item, called a
# key.  The second specifies its value.
$VarSetIn.put("Client.Version.Build", [long]0)
$VarSetIn.put("Client.Version.Major", [long]8)
$VarSetIn.put("Client.Version.Minor", [long]70)
$VarSetIn.put("Client.Version.Release", [long]696)
$VarSetIn.put("LocaleID", [long]1033)
$VarSetIn.put("OperationName", "UserGetInfo")
$VarSetIn.put("Properties.$McsFriendlyName", "")
$VarSetIn.put("Properties.$McsFriendlyPath", "")
$VarSetIn.put("Properties.$McsLocalAccount", "")
$VarSetIn.put("Properties.AccountDisabled", "")
$VarSetIn.put("Properties.AccountExpirationDate", "")
$VarSetIn.put("Properties.IsAccountLocked", "")
$VarSetIn.put("Properties.displayName", "")
$VarSetIn.put("Properties.manager", "")
$VarSetIn.put("Properties.sAMAccountName", "")
$VarSetIn.put("Properties.userPrincipalName", "")
```

```
$VarSetIn.put("User", "OnePoint://CN=Bob Slydell,CN=Users,DC=nqtraining,DC=lab")
$VarSetIn.put("VisibleProperties", $true)
#Submit the varset
$vsOut = $gEaServerObject.ScriptSubmit($VarSetIn)
if ($vsOut.get("Errors.numErrors") -gt 0) {
    write-host $vsOut.get("Errors.numErrors") " errors occurred!"
    return
}
$vsOut.get("Properties.sAMAccountName")
$vsOut.get("Properties.userPrincipalName")
$vsOut.get("Properties.displayname")
$vsOut.get("Properties.IsAccountLocked")
```

As with earlier DRA releases, the `CreateScriptsExt.dll` extension can be copied to the \NetIQ\DRA folder and registered by an Administrator account on 64-bit platforms. You can select Varset text lines and use the extension appearing in the **Tools** menu to create VBScript text. That text can be transformed to PowerShell by using the conventions shown in the example above.

# Legal Notice

For information about legal notices, trademarks, disclaimers, warranties, export and other use restrictions, U.S. Government rights, patent policy, and FIPS compliance, see https://www.microfocus.com/about/legal/.

**© Copyright 2007 – 2020 Micro Focus or one of its affiliates.**

The only warranties for products and services of Micro Focus and its affiliates and licensors (“Micro Focus”) are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. Micro Focus shall not be liable for technical or editorial errors or omissions contained herein. The information contained herein is subject to change without notice.