Chapter 12 Lab

We bet you can guess what is coming. You’ll be adding type information and creating custom format files for the functions you’ve been working on the last several chapters. Use the dotnettypes.format.ps1xml and other .ps1xml files as sources for sample layout. Copy and paste the XML into your new format file. Don’t forget that tags are case-sensitive.

* + 1. Lab A

Modify your advanced function from Lab A in Chapter 10 so that the output object has the type name MOL.ComputerSystemInfo. Then, create a custom view in a file named C:\CustomViewA.format.ps1xml. The custom view should display objects of the type MOL.ComputerSystemInfo in a list format, displaying the information in a list as indicated in your design for this lab. Go back to Chapter 6 to check what the output names should be.

At the bottom of the script file, add these commands to test:

Update-FormatData –prepend c:\CustomViewA.format.ps1xml

<function-name> -ComputerName localhost

The final output should look something like the following.

Computername : CLIENT2

Workgroup :

AdminPassword : NA

Model : VirtualBox

Manufacturer : innotek GmbH

BIOSSerialNumber : 0

OSVersion : 6.1.7601

SPVersion : 1

Note that the list labels are not exactly the same as the custom object’s property names.

Sample format file

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

<Configuration>

<ViewDefinitions>

<View>

<Name>MOL.SystemInfo</Name>

<ViewSelectedBy>

<TypeName>MOL.ComputerSystemInfo</TypeName>

</ViewSelectedBy>

<ListControl>

<ListEntries>

<ListEntry>

<ListItems>

<ListItem>

<PropertyName>ComputerName</PropertyName>

</ListItem>

<ListItem>

<PropertyName>Workgroup</PropertyName>

</ListItem>

<ListItem>

<PropertyName>AdminPassword</PropertyName>

</ListItem>

<ListItem>

<Propertyname>Model</Propertyname>

</ListItem>

<ListItem>

<Propertyname>Manufacturer</Propertyname>

</ListItem>

<ListItem>

<Propertyname>SerialNumber</Propertyname>

<Label>BIOSSerialNumber</Label>

</ListItem>

<ListItem>

<Propertyname>Version</Propertyname>

<Label>OSVersion</Label>

</ListItem>

<ListItem>

<Propertyname>ServicePackMajorVersion</Propertyname>

<Label>SPVersion</Label>

</ListItem>

</ListItems>

</ListEntry>

</ListEntries>

</ListControl>

</View>

</ViewDefinitions>

</Configuration>

Sample Script

Function Get-ComputerData {

<#

.SYNOPSIS

Get computer related data

.DESCRIPTION

This command will query a remote computer and return a custom object

with system information pulled from WMI. Depending on the computer

some information may not be available.

.PARAMETER Computername

The name of a computer to query. The account you use to run this function

should have admin rights on that computer.

.PARAMETER ErrorLog

Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE

PS C:\> Get-ComputerData Server01

Run the command and query Server01.

.EXAMPLE

PS C:\> get-content c:\work\computers.txt | Get-ComputerData -Errorlog c:\logs\errors.txt

This expression will go through a list of computernames and pipe each name

to the command. Computernames that can't be accessed will be written to

the log file.

#>

[cmdletbinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[ValidateNotNullorEmpty()]

[string[]]$ComputerName,

[string]$ErrorLog="C:\Errors.txt"

)

Begin {

Write-Verbose "Starting Get-Computerdata"

}

Process {

foreach ($computer in $computerName) {

Write-Verbose "Getting data from $computer"

Try {

Write-Verbose "Win32\_Computersystem"

$cs = Get-WmiObject -Class Win32\_Computersystem -ComputerName $Computer -ErrorAction Stop

#decode the admin password status

Switch ($cs.AdminPasswordStatus) {

1 { $aps="Disabled" }

2 { $aps="Enabled" }

3 { $aps="NA" }

4 { $aps="Unknown" }

}

#Define a hashtable to be used for property names and values

$hash=@{

Computername=$cs.Name

Workgroup=$cs.WorkGroup

AdminPassword=$aps

Model=$cs.Model

Manufacturer=$cs.Manufacturer

}

} #Try

Catch {

#create an error message

$msg="Failed getting system information from $computer. $($\_.Exception.Message)"

Write-Error $msg

Write-Verbose "Logging errors to $errorlog"

$computer | Out-File -FilePath $Errorlog -append

} #Catch

#if there were no errors then $hash will exist and we can continue and assume

#all other WMI queries will work without error

If ($hash) {

Write-Verbose "Win32\_Bios"

$bios = Get-WmiObject -Class Win32\_Bios -ComputerName $Computer

$hash.Add("SerialNumber",$bios.SerialNumber)

Write-Verbose "Win32\_OperatingSystem"

$os = Get-WmiObject -Class Win32\_OperatingSystem -ComputerName $Computer

$hash.Add("Version",$os.Version)

$hash.Add("ServicePackMajorVersion",$os.ServicePackMajorVersion)

#create a custom object from the hash table

$obj=New-Object -TypeName PSObject -Property $hash

#add a type name to the custom object

$obj.PSObject.TypeNames.Insert(0,'MOL.ComputerSystemInfo')

Write-Output $obj

#remove $hash so it isn't accidentally re-used by a computer that causes

#an error

Remove-Variable -name hash

} #if $hash

} #foreach

} #process

End {

Write-Verbose "Ending Get-Computerdata"

}

}

Update-FormatData –prepend C:\CustomViewA.format.ps1xml

Get-ComputerData -ComputerName localhost

* + 1. Lab B

Modify your advanced function Lab B from Chapter 10 so that the output object has the type name MOL.DiskInfo. Then, create a custom view in a file named C:\CustomViewB.format.ps1xml. The custom view should display objects of the type MOL.DiskInfo in a table format, displaying the information in a table as indicated in your design for this lab. Refer back to Chapter 6 for a refresher. The column headers for the FreeSpace and Size properties should display “FreeSpace(GB)” and “Size(GB),” respectively.

At the bottom of the script file, add these commands to test:

Update-FormatData –prepend c:\CustomViewB.format.ps1xml

<function-name> -ComputerName localhost

The final output should look something like the following.

ComputerName Drive FreeSpace(GB) Size(GB)

------------ ----- ------------- --------

CLIENT2 \\?\Volume{8130d5f3-8e9b-... 0.07 0.10

CLIENT2 C:\Temp\ 9.78 10.00

CLIENT2 C:\ 2.72 19.90

CLIENT2 D:\ 2.72 4.00

Note that the column headers are not exactly the same as the custom object’s property names.

Sample format file solution

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

<Configuration>

<ViewDefinitions>

<View>

<Name>MOL.SystemInfo</Name>

<ViewSelectedBy>

<TypeName>MOL.DiskInfo</TypeName>

</ViewSelectedBy>

<TableControl>

<TableHeaders>

<TableColumnHeader>

<Width>18</Width>

</TableColumnHeader>

<TableColumnHeader/>

<TableColumnHeader>

<Label>FreeSpace(GB)</Label>

<Width>15</Width>

</TableColumnHeader>

<TableColumnHeader>

<Label>Size(GB)</Label>

<Width>10</Width>

</TableColumnHeader>

</TableHeaders>

<TableRowEntries>

<TableRowEntry>

<TableColumnItems>

<TableColumnItem>

<PropertyName>ComputerName</PropertyName>

</TableColumnItem>

<TableColumnItem>

<PropertyName>Drive</PropertyName>

</TableColumnItem>

<TableColumnItem>

<PropertyName>FreeSpace</PropertyName>

</TableColumnItem>

<TableColumnItem>

<Propertyname>Size</Propertyname>

</TableColumnItem>

</TableColumnItems>

</TableRowEntry>

</TableRowEntries>

</TableControl>

</View>

</ViewDefinitions>

</Configuration>

Sample script solution

Function Get-VolumeInfo {

<#

.SYNOPSIS

Get information about fixed volumes

.DESCRIPTION

This command will query a remote computer and return information about fixed

volumes. The function will ignore network, optical and other removable drives.

.PARAMETER Computername

The name of a computer to query. The account you use to run this function

should have admin rights on that computer.

.PARAMETER ErrorLog

Specify a path to a file to log errors. The default is C:\Errors.txt

.EXAMPLE

PS C:\> Get-VolumeInfo Server01

Run the command and query Server01.

.EXAMPLE

PS C:\> get-content c:\work\computers.txt | Get-VolumeInfo -errorlog c:\logs\errors.txt

This expression will go through a list of computernames and pipe each name

to the command. Computernames that can't be accessed will be written to

the log file.

#>

[cmdletbinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[ValidateNotNullorEmpty()]

[string[]]$ComputerName,

[string]$ErrorLog="C:\Errors.txt",

[switch]$LogErrors

)

Begin {

Write-Verbose "Starting Get-VolumeInfo"

}

Process {

foreach ($computer in $computerName) {

Write-Verbose "Getting data from $computer"

Try {

$data = Get-WmiObject -Class Win32\_Volume -computername $Computer -Filter "DriveType=3" -ErrorAction Stop

Foreach ($drive in $data) {

Write-Verbose "Processing volume $($drive.name)"

#format size and freespace

$Size="{0:N2}" -f ($drive.capacity/1GB)

$Freespace="{0:N2}" -f ($drive.Freespace/1GB)

#Define a hashtable to be used for property names and values

$hash=@{

Computername=$drive.SystemName

Drive=$drive.Name

FreeSpace=$Freespace

Size=$Size

}

#create a custom object from the hash table

$obj=New-Object -TypeName PSObject -Property $hash

#Add a type name to the object

$obj.PSObject.TypeNames.Insert(0,'MOL.DiskInfo')

Write-Output $obj

} #foreach

#clear $data for next computer

Remove-Variable -Name data

} #Try

Catch {

#create an error message

$msg="Failed to get volume information from $computer. $($\_.Exception.Message)"

Write-Error $msg

Write-Verbose "Logging errors to $errorlog"

$computer | Out-File -FilePath $Errorlog -append

}

} #foreach computer

} #Process

End {

Write-Verbose "Ending Get-VolumeInfo"

}

}

Update-FormatData –prepend C:\CustomViewB.format.ps1xml

Get-VolumeInfo localhost

* + 1. Lab C

Modify your advanced function Lab C from Chapter 10 so that the output object has the type name MOL.ServiceProcessInfo. Then, create a custom view in a file named C:\CustomViewC.format.ps1xml. The custom view should display objects of the type MOL.ServiceProcessInfo in a table format, displaying computername, service name, display name, process name, and process virtual size.

In addition to the table format, create a list view in the same file that displays the properties in this order:

* Computername
* Name (renamed as Service)
* Displayname
* ProcessName
* VMSize
* ThreadCount
* PeakPageFile

At the bottom of the script file, add these commands to test:

Update-FormatData –prepend c:\CustomViewC.format.ps1xml

<function-name> -ComputerName localhost

<function-name> -ComputerName localhost | Format-List

The final output should look something like this for the table.

ComputerName Service Displayname ProcessName VM

------------ ------- ----------- ----------- --

CLIENT2 AudioEndpo... Windows Audio E... svchost.exe 172208128

CLIENT2 BFE Base Filtering ... svchost.exe 69496832

CLIENT2 BITS Background Inte... svchost.exe 499310592

CLIENT2 Browser Computer Browser svchost.exe 499310592

And like this for the list:

Computername : CLIENT2

Service : AudioEndpointBuilder

Displayname : Windows Audio Endpoint Builder

ProcessName : svchost.exe

VMSize : 172208128

ThreadCount : 13

PeakPageFile : 83112

Note that per the design specifications from Chapter 6 not every object property is displayed by default and that some column headings are different than the actual property names.

Sample format file solution:

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

<Configuration>

<ViewDefinitions>

<View>

<Name>MOL.SystemInfo</Name>

<ViewSelectedBy>

<TypeName>MOL.ServiceProcessInfo</TypeName>

</ViewSelectedBy>

<TableControl>

<TableHeaders>

<TableColumnHeader>

<Width>14</Width>

</TableColumnHeader>

<TableColumnHeader>

<Label>Service</Label>

<Width>13</Width>

</TableColumnHeader>

<TableColumnHeader>

<Width>18</Width>

</TableColumnHeader>

<TableColumnHeader>

<Width>17</Width>

</TableColumnHeader>

<TableColumnHeader>

<Label>VM</Label>

<Width>14</Width>

</TableColumnHeader>

</TableHeaders>

<TableRowEntries>

<TableRowEntry>

<TableColumnItems>

<TableColumnItem>

<PropertyName>ComputerName</PropertyName>

</TableColumnItem>

<TableColumnItem>

<PropertyName>Name</PropertyName>

</TableColumnItem>

<TableColumnItem>

<PropertyName>Displayname</PropertyName>

</TableColumnItem>

<TableColumnItem>

<Propertyname>ProcessName</Propertyname>

</TableColumnItem>

<TableColumnItem>

<Propertyname>VMSize</Propertyname>

</TableColumnItem>

</TableColumnItems>

</TableRowEntry>

</TableRowEntries>

</TableControl>

</View>

<View>

<Name>MOL.SystemInfo</Name>

<ViewSelectedBy>

<TypeName>MOL.ServiceProcessInfo</TypeName>

</ViewSelectedBy>

<ListControl>

<ListEntries>

<ListEntry>

<ListItems>

<ListItem>

<PropertyName>ComputerName</PropertyName>

</ListItem>

<ListItem>

<PropertyName>Name</PropertyName>

<Label>Service</Label>

</ListItem>

<ListItem>

<PropertyName>Displayname</PropertyName>

</ListItem>

<ListItem>

<Propertyname>ProcessName</Propertyname>

</ListItem>

<ListItem>

<Propertyname>VMSize</Propertyname>

</ListItem>

<ListItem>

<Propertyname>ThreadCount</Propertyname>

</ListItem>

<ListItem>

<Propertyname>PeakPageFile</Propertyname>

</ListItem>

</ListItems>

</ListEntry>

</ListEntries>

</ListControl>

</View>

</ViewDefinitions>

</Configuration>

Sample script solution:

Function Get-ServiceInfo {

<#

.SYNOPSIS

Get service information

.DESCRIPTION

This command will query a remote computer for running services and write

a custom object to the pipeline that includes service details as well as

a few key properties from the associated process. You must run this command

with credentials that have admin rights on any remote computers.

.PARAMETER Computername

The name of a computer to query. The account you use to run this function

should have admin rights on that computer.

.PARAMETER ErrorLog

Specify a path to a file to log errors. The default is C:\Errors.txt

.PARAMETER LogErrors

If specified, computer names that can't be accessed will be logged

to the file specified by -Errorlog.

.EXAMPLE

PS C:\> Get-ServiceInfo Server01

Run the command and query Server01.

.EXAMPLE

PS C:\> get-content c:\work\computers.txt | Get-ServiceInfo -logerrors

This expression will go through a list of computernames and pipe each name

to the command. Computernames that can't be accessed will be written to

the log file.

#>

[cmdletbinding()]

param(

[Parameter(Position=0,ValueFromPipeline=$True)]

[ValidateNotNullorEmpty()]

[string[]]$ComputerName,

[string]$ErrorLog="C:\Errors.txt",

[switch]$LogErrors

)

Begin {

Write-Verbose "Starting Get-ServiceInfo"

#if -LogErrors and error log exists, delete it.

if ( (Test-Path -path $errorLog) -AND $LogErrors) {

Write-Verbose "Removing $errorlog"

Remove-Item $errorlog

}

}

Process {

foreach ($computer in $computerName) {

Write-Verbose "Getting services from $computer"

Try {

$data = Get-WmiObject -Class Win32\_Service -computername $Computer -Filter "State='Running'" -ErrorAction Stop

foreach ($service in $data) {

Write-Verbose "Processing service $($service.name)"

$hash=@{

Computername=$data[0].Systemname

Name=$service.name

Displayname=$service.DisplayName

}

#get the associated process

Write-Verbose "Getting process for $($service.name)"

$process=Get-WMIObject -class Win32\_Process -computername $Computer -Filter "ProcessID='$($service.processid)'" -ErrorAction Stop

$hash.Add("ProcessName",$process.name)

$hash.add("VMSize",$process.VirtualSize)

$hash.Add("PeakPageFile",$process.PeakPageFileUsage)

$hash.add("ThreadCount",$process.Threadcount)

#create a custom object from the hash table

$obj=New-Object -TypeName PSObject -Property $hash

#add a type name to the custom object

$obj.PSObject.TypeNames.Insert(0,'MOL.ServiceProcessInfo')

Write-Output $obj

} #foreach service

}

Catch {

#create an error message

$msg="Failed to get service data from $computer. $($\_.Exception.Message)"

Write-Error $msg

if ($LogErrors) {

Write-Verbose "Logging errors to $errorlog"

$computer | Out-File -FilePath $Errorlog -append

}

}

} #foreach computer

} #process

End {

Write-Verbose "Ending Get-ServiceInfo"

}

}

Update-FormatData –prepend C:\CustomViewC.format.ps1xml

Get-ServiceInfo -ComputerName "localhost"

Get-ServiceInfo -ComputerName "localhost" | format-list