Technical Report

NetApp PowerShell Survival Guide

Chris Lionetti, NetApp
February 2011| TR-3896

TABLE OF CONTENTS

1 INTRODUCTION AND INSTALLATION

Microsoft® Windows® PowerShell is a task automation framework that consists of a command-line interface (CLI) shell and an associated scripting language built on top of, and integrated with, the Microsoft.NET Framework.[1] PowerShell enables administrators to perform administrative tasks on both local and remote Windows systems.

PowerShell works equally well as both a CLI and as a fully functional scripting language. The language is user friendly, and it is also very powerful. Even without using any of its advanced language features, PowerShell is a direct replacement for the functionality that Telnet offers to the storage controller. PowerShell has an easy learning curve, and it should take less than one hour to become fluent in using it as a Telnet replacement. If you learn the advanced features, which only takes a little bit longer, you can solve more complex situations and increase automation.

1.1 INTENDED AUDIENCE

This guide is intended for the overworked storage administrator or storage engineer. In general, the intended audience is first-time users of PowerShell. Administrators who are currently deploying NetApp® storage for a Windows Server 2008 R2 environment and need a higher level of efficiency and automation will gain the most benefit from this guide.

No assumptions are made about the reader's ability to program. If you are a serious programmer, this guide is probably too basic for your needs. However, even serious programmers can benefit from the discussion of Data ONTAP® concepts in this guide.

1.2 DETERMINE WHICH VERSION OF POWERSHELL IS RUNNING

PowerShell version 2.0 must be used to take full advantage of the Data ONTAP PowerShell Module. To find out which version of PowerShell is installed, execute the following command:

```
PowerShell Prompt> $host.version
```

Note: Refer to section 1.3 for directions on how to launch a PowerShell prompt to determine which version is installed.

The output of this command shows which major version is installed—either version 1.0 or 2.0.

Table 1 shows which operating system comes with each version of PowerShell. If you are running a version previous to 2.0, then download PowerShell Version 2.0 directly from the Microsoft Web site.

Table 1) Operating systems for each PowerShell version.

Operating System	Version	Delivery Method
Windows Server 2008 R2	Version 2.0	Installed by default
Windows Server 2008 R2 Core	Version 2.0	Installed by default
Windows 7	Version 2.0	Installed by default
Windows Server 2008	Version 1.0	Not installed by default

[1] Wikipedia, Windows Power Shell; http://en.wikipedia.org/wiki/Windows_PowerShell; accessed 12/6/2010.

Operating System	Version	Delivery Method
Windows Server 2008 Core	Not supported	Not supported
Windows Vista®	Version 1.0	Not installed by default
Windows Server 2003	None	Must download
Windows XP and earlier	None	Must download

Note: Microsoft.NET Framework 2.0 is a prerequisite of PowerShell. Therefore, installing PowerShell 2.0 on an operating system such as XP requires both products to be installed. The Microsoft.NET Framework is not permitted on Windows Server Core 2008; therefore, PowerShell cannot be loaded on Windows Server Core 2008.

1.3 LAUNCH A POWERSHELL COMMAND WINDOW

To launch a PowerShell CLI screen on newer operating systems, such as Windows Server 2008 R2 and Windows 7:

- Click the PowerShell icon on the start bar.

Or complete the following steps:

1. Click Start.
2. Click Administrator Tools.
3. Click PowerShell.

Note: For earlier versions of Microsoft Windows that require a PowerShell download, check for the default location under the Start menu.

The PowerShell Command window can be launched using three different methods, all of which can be accessed by right-clicking the PowerShell icon on the Start bar. The three options are as follows:

- Select the Run As Current user standard PowerShell Command window.
- Select Load All Current Modules, which automatically loads all available modules.
- Select the PowerShell ISE debugging/scripting environment.

Note: The directions in the remainder of this document are based on the assumption that PowerShell was launched by selecting the Load All Current Modules option so that all modules are loaded.

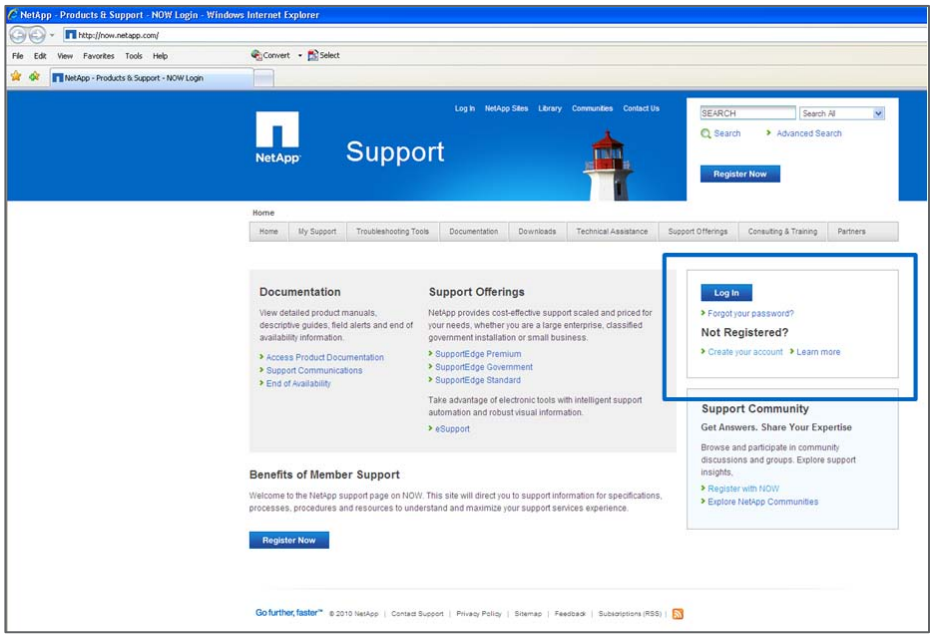
1.4 DOWNLOAD THE NETAPP DATA ONTAP POWERSHELL LIBRARY

To download the NetApp Data ONTAP PowerShell Library, you need a NOW™ (NetApp on the Web) account and a Web browser. Complete the following steps to download the Data ONTAP PowerShell library:

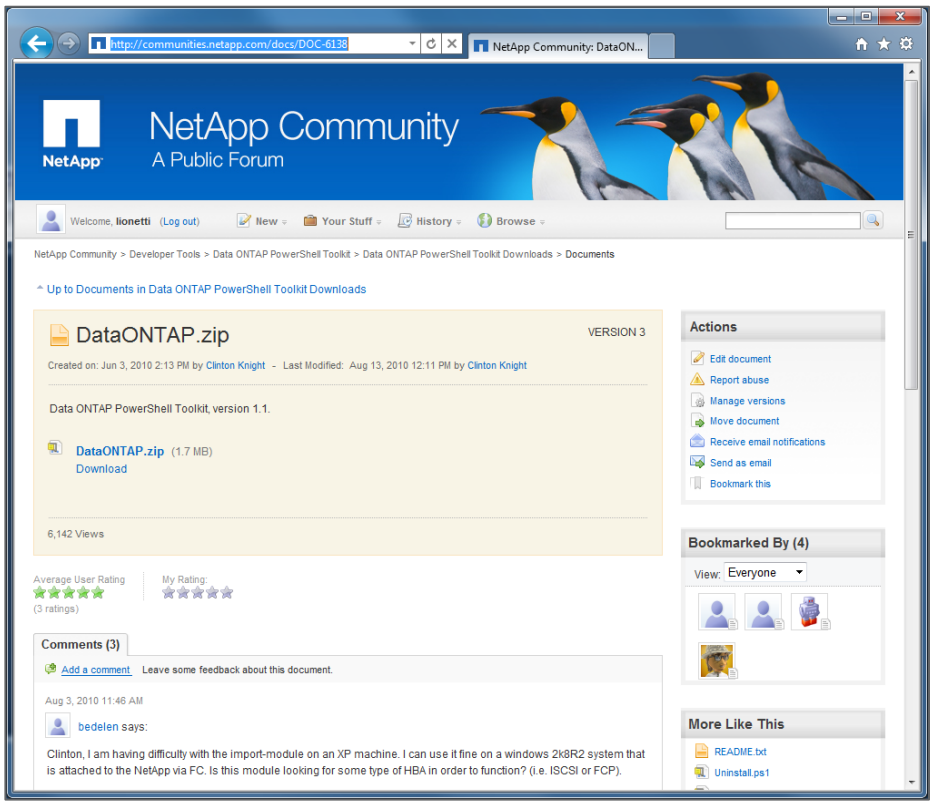
1. Create a NOW account if needed.
   a. Get your NetApp serial number.
   b. Click the Create your account hyperlink on the NOW Home screen (as shown in Figure 1).

Note: The Communities and Forums Web pages that contain samples of PowerShell code are available without logging into the NOW Web site. However, the Module download does not display unless you are logged in.

Figure 1) Create an account on the NOW Web site.



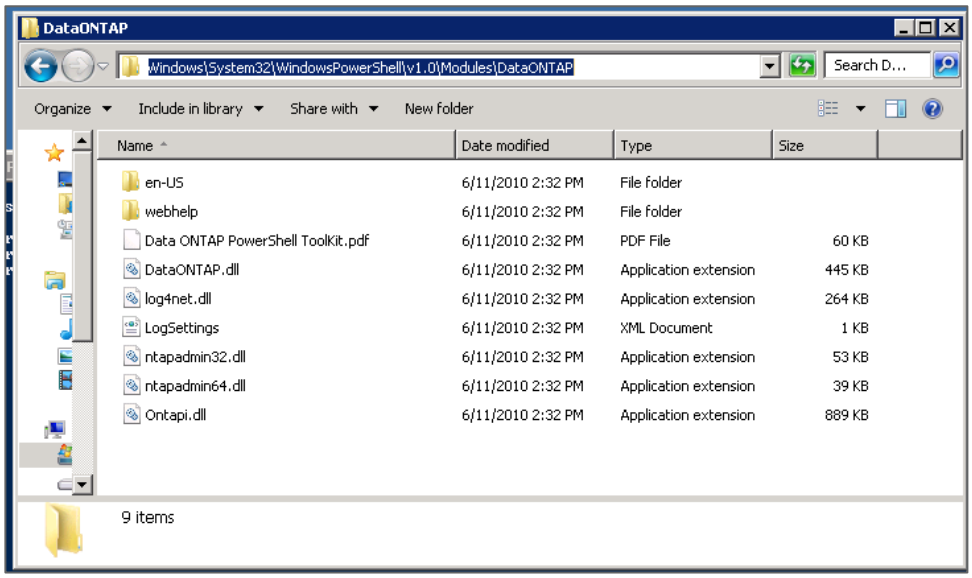
2. Enter the following Web site in the address window http://communities.netapp.com/docs/DOC-6138 (after you have created an account and logged in).



3. Download the DataONTAP.ZIP file to a temporary directory on your machine.
4. Extract the file.
5. Place the Module contents in the following location on your Windows machine: `C:\Windows\System32\WindowsPowerShell\v1.0\Modules\DataOntap`.

Note: The extracted files must be in exactly the right place. The files such as `ontapi.dll` must not be one level deeper in the directory tree structure. Figure 2 shows where the extracted files should be located so that PowerShell can find these modules.

Figure 2) Put extracted files in the right place.



1.5 LOAD THE MODULE IN POWERSHELL

To import modules manually from the basic-level PowerShell prompt, complete the following steps:

1. Execute the following commands (refer to section 1.4 for directions on how and where to install the module files):

```
PowerShell Prompt> Set-ExecutionPolicy unrestricted
PowerShell Prompt> Import-module DataOnTap
```

2. Verify that the Data ONTAP Module is loaded by executing the following command:

```
PowerShell Prompt> get-module
```

Note: This command shows which modules are currently installed.

Other useful modules can be installed and downloaded from Web sites such as www.CodePlex.com for modules such as the Hyper-V™ PowerShell module, which is used later in this document in a few sample scripts. Modules might also install automatically when you add features to a Windows 2008 R2 server.

For example, to run commands to modify a Windows 2008 R2 Failover Cluster from your Microsoft System Center Virtual Machine Manager (SCVMM) server, install the Windows Failover Cluster feature on the SCVMM server. The feature does not need to be configured or enabled on the SCVMM server; it is simply available to manage clusters remotely. The PowerShell Module will be installed and functional from that point on.

2 FIRST POWERSHELL COMMAND

PowerShell commands can be used interactively similar to the way Telnet or RSH commands operate. In this mode, after typing a command at the command prompt, you get immediate feedback on the results. This section covers PowerShell as a direct replacement in this type of interactive environment. The following sections cover how to use pipes to make commands smarter and how to use scripts to automate the infrastructure.

The very first command to run is the command to connect to the controller itself. This command uses as an argument the IP address of the array as well as the user name to connect with; for example:

```
PowerShell Prompt> Connect-NAController 10.58.99.254 –cred administrator
```

This command tells PowerShell to connect to the controller with the IP address 10.58.99.254 and to use the credentials of Administrator. A pop-up box displays that asks for the password. After the password is entered, the command returns information regarding the connection, such as the connection type.

2.1 POWERSHELL COMMAND NAMING CONVENTION

PowerShell commands follow a naming convention of “verb-noun” to describe what action to complete on which device. Common command verbs and nouns are shown in Table 2.

Table 2) Command verbs and nouns.

Common Command Verbs	Common Command Nouns
Get	Aggr
Set	Vol
Invoke	Controller
New	SnapMirror
Add	SIS
Enable	Igroup
Remove	

To obtain the full list of the commands that are supplied in the Data ONTAP toolkit, complete the following steps:

1. Type the following command:

```
PowerShell Prompt> Show-naHelp
```

2. Click on the Commandlets tab.

2.2 POWERSHELL ORDER OF ARGUMENTS

Each command listed shows a list of possible arguments, but also makes several assumptions. The command help for the **get-NaLun** command with possible options is shown in the following console box.

```
New-NaLun [-Path] <String> [-Type <String>] [-Unreserved] [-Size] <String> [
    -Controller <NaController>] [-WarningAction <ActionPreference>] [-WarningVar
    iable <String>] [-WhatIf] [-Confirm] [<CommonParameters>]
```

In this command:

- `[-Path]<String>` shows that the identifier of `–Path` is optional, but it assumes that the first nondefined item you pass will be a path.
- The second string shows that it expects to be assigned to the Size Argument.

Many of the **`Get`** commands have the simplest version with no arguments, and return a list of ALL of the objects defined. Some of the arguments are optional and use default behavior. If you do not specify a controller to run the command against, the command runs against the last controller that was connected by using the **`Connect-NAController`** command.

The other assumption that PowerShell makes is that if you don't specify the order of arguments that you pass to a PowerShell command, they will be in the order specified by the command. For example, if the following command is sent to the controller:

```
PowerShell Prompt> New-NaLun /vol/vol3/data1 10000000
```

PowerShell creates a logical unit number (LUN) called `/vol/vol3/data1` that is 10,000,000 bytes in size. Arguments can be passed to PowerShell commands in any order if the fields in which each argument is destined to be inserted are specified. The following command, even if it's out of order, works properly:

```
PowerShell Prompt> New-NaLun –path /vol/vol3/data1 –size 1000000
```

Alternatively, the commands can be sent in the correct order without being forced to use any argument qualifier, as the following example illustrates. In the following examples, a bold font is used for all information typed in by the user.

```
Telnet 10.58.96.222
Username: administrator
Password: password
Telnet> lun show
      /vol/CSV2/CSV2PRI          500.1g (536952700928)(r/w, online, mapped)
      /vol/CSV3/CSV3PRI          90.0g (96638814720)(r/o, online)
      /vol/SDELETE/test1         10.0g (10742215680)(r/w, online, mapped)
      /vol/SEARCSV1/CSV1         600.1g (644335534080)(r/w, online, mapped)
      /vol/SEARCSV1/Q            1.0g (1077511680)(r/w, online, mapped)
      /vol/SEARCSV1/SCOMSCVMM    75.0g (80541941760)(r/w, online, mapped)
      /vol/SQLInstance/SQLMOUNT  3.0g (3224309760)(r/w, online)
      /vol/SQLInstance/sqldata   75.0g (80541941760)(r/w, online)
      /vol/SQLInstance/sqldata1  288.2g (309435033600)(r/w, online, mapped)
      /vol/SQLInstance/sqldata2  50.0g (53694627840)(r/w, online, mapped)
      /vol/SQLInstance/sqllog    10g (10737418240)(r/w, online)
Telnet>
```

As a comparison, the command to list all LUNs from the Telnet Screen would look like this:

```
PowerShell Prompt > Connect-NaController 10.58.96.222 -cred administrator
Name     Address      Ontapi   Version
----     -------      ------   -------

10.58.96.222         10.58.96.222      1.12
PowerShell Prompt > get-NaLun

Path                        Protocol     Online  Mapped  Thin   Comment
----                        ---------    ------- ------  -----  -------
/vol/chmcclin_clone/lun_2  windows_2008   False  False   True
/vol/chmcclin_base/lun_2   windows_2008   True   False   True
/vol/chmcclin_base/lun_1   windows_2008   True   False   True
/vol/vol2/readonly2        hyper_v        True   True    True
/vol/sdw_cl_vol2_0/ISO     windows_2008   False  False   True   Windows ISO CD Images
/vol/sdw_cl_vol2_0/readonlytest   hyper_v        True   True    True
/vol/vol2/readonlytest     hyper_v        True   False   True
/vol/SDELETE/test1         windows_2008   True   True    True   TestDeDupe
/vol/vol1/DynTest1         hyper_v        True   True    True
/vol/SQLInstance/SQLMOUNT  windows        True   False   True   SQL Root Mount Point
PowerShell Prompt >
```

As you can see from the output, the steps to get information directly from PowerShell have the same complexity as the steps from the Telnet prompt. In fact, the advantage of better formatted output is immediately apparent.

Another major advantage of PowerShell over Telnet is that Telnet has a limited number of active connections to the array, whereas with PowerShell you can connect from multiple places at the same time.

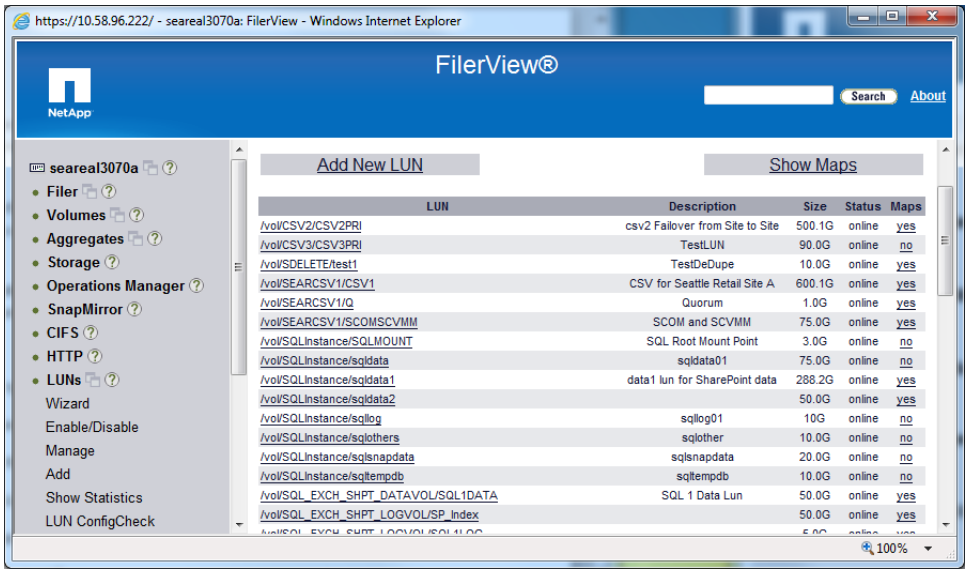
For completeness, this section also includes the GUI window to an array to show the same types of information that one would get from either NetApp FilerView® or NetApp System Manager.

To get information from FilerView, complete the following steps:

1. Open a Web browser
2. Type `HTTP://10.58.96.222/na_admin` in the address bar.
3. Enter your user name, which is administrator, and your password into the log-in box.
4. Click on FilerView to open the FilerView screen from the Administrative Overview window.
5. Click on LUNs to open the LUN drop-down list.
6. Click on Manage LUNs to open the screen that displays the LUN data.

A detailed screen is shown in Figure 3.

Figure 3) LUN data displayed on FilerView screen.



2.3 POWERSHELL COMMON COMMANDS

The following section covers the most commonly used commands and discusses the basic prerequisites for each command. The commands listed here don't show all of the possible subcommands, but only represent the commands required to perform the basic duties. PowerShell shorthand is used for these cases, and the values are fed into the command in the expected order. For example, the following two commands do the same thing: The first example is in shorthand, and the other is spelled out fully.

```
PowerShell Prompt> New-NaAggr Aggr1 5 73GB
PowerShell Prompt> New-NaAggr –name Aggr1 –diskcount 5 –disksize 73GB
```

3 HOW TO CHANGE FROM CLI TO SCRIPT

The good news is that the same exact commands used to run CLI commands can be saved with the extension of `.ps1` and can be directly run as PowerShell scripts. If you are unsure of how a single line works, you can type in that line directly from the command prompt to test it.

When running scripts, you must get used to one unusual step: You must specify the directory path from which to run the script. That is, if you are in the `C:\Software\` directory and you type `MYSCRIPT.ps1`, PowerShell will fail. You must type `.\MYSCRIPT.ps1` so that PowerShell knows you expect to find the script in the current directory.

3.1 USE VARIABLES

Variables can store commonly used values and provide access to those values later in subsequent commands. For example, a set of variables will speed up the scripting work if you:

- Have a set of three servers.
- Know the Igroup for each of those servers' HBAs.
- Commonly call commands to map LUNs to them.

For example, you could create a variable block at the start of each script as shown in the next example.

The rest of the script can then call on and use these variables at will. The following console box shows the remainder of the script for the following scenario:

1. Create three new LUNs.
2. Assign each LUN to a different server.
3. Assign the third LUN to a two-node cluster.

Note: The # sign is used to denote comments instead of code. All comments within a script are preceded by the # sign.

```
#start variable block
$Node1 = "TK2WUSWDSDC1"
$Node2 = "TK2HYPVN1"
$Node3 = "TK2HYPVN2"
# End variable block
Connect-NAController 10.58.99.254 –cred administrator
New-NaLun /vol/Infra/WDSData1 500000000000 –type windows_2008 -unreserved
New-NaLun /vol/VMSet1/VM36 100000000000 –type windows_2008 –unreserved
Add-NaLunMap /vol/infra/WDSData1 $Node1
Add-NaLunMap /vol/VMSet1/VM36 $Node2
Add-NaLunMap /vol/VMSet1/VM36 $Node3
```

3.2 POPULATE VARIABLES FROM COMMANDS

You can also run a **`Get`** command and insert the results of that command into a variable. However, the results from a command are not usually simple yes/no answers, but are more commonly sets of data. When more than a single thing is returned from a result, then the entire set of data (called an object) is placed into that variable.

- To get a few key points of data about the vol0 volume, execute the following command:

```
PowerShell Prompt> Get-NaVol vol0
```

- From the command prompt, execute the following command to populate the `$Temp` variable:

```
PowerShell Prompt> $Temp=Get-NaVol vol0
```

PowerShell returns right back to the prompt.

- To tell PowerShell to output the results of the `$temp` variable, type in the variable name and press Enter:

```
PowerShell Prompt> $Temp
```

- Use `dot notation` to get to any individual field inside that object. For example, to only return to the State of that volume, execute the following command:

```
PowerShell Prompt> $Temp.State
```

The returned value should come back as online.

You can use the command **`get-NaLun`** to return a collection of objects. In this case, each LUN is an object with many properties, but these objects can have hundreds of LUNs. In this case, the variable becomes a collection.

- Execute the following commands to get into the collection:

```
PowerShell Prompt> $Temp=Get-NaLun
PowerShell Prompt> $Temp[3].Path
PowerShell Prompt> $Temp[3].Protocol
```

These commands give the path and then the protocol that is used for the fourth LUN in the collection (numbering starts at zero). As usual, if you only type `$temp` and press `Enter`, the output will be a flood of information that looks identical to running the command.

3.3 POPULATE A COLLECTION

PowerShell includes a concept called a collection. A collection is actually a collection of common variables. The variables can be defined as follows:

```
$HostNames = @("host2","Host3","Host4")
```

The collection can be used as a set, and it presents a much simpler way to chain operations that you want to happen. Refer to section 3.6 for an example of how to walk through a collection.

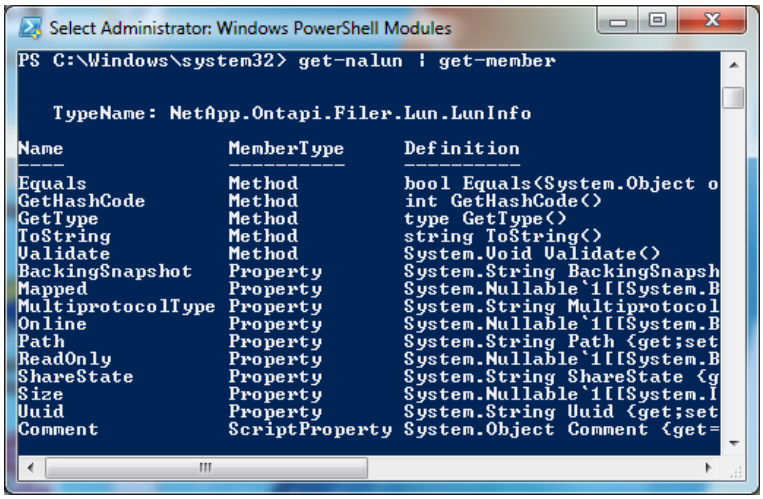
3.4 FIND ALL THE METHODS OF AN OBJECT

To find out what methods are available for each command, execute the following command:

```
PowerShell Prompt> Get-NaLun | Get-Member
```

This command exposes all of these subvariables for each object. The output lists the names of all of the subcommands. For example, the variables that are available for the **`Get-nalun`** command are shown in Figure 4.

Figure 4) Variables available for get-nalun command.



Note: In Figure 4, the Member Types that we care about are all listed as Property, and the Member Types listed as Method are common among all PowerShell objects.

3.5 USE IF STATEMENTS

Automation usually requires that actions happen only under specific conditions, and these specific conditions are tested using common `If` statements. For example:

- Create a new LUN in a volume only if the volume has enough free space; if it does not, create that LUN in another volume.

The compare operator, `-gt`, is inside the `If` statement. This `-gt` operator means "is greater than." You can nest many `If` functions within each other as well as use "Else" or "Elseif" statements.

```
$TargetVol = “/vol/Vol7”
$TargetLUN = “/vol/Vol7/LUN31”
$LunSize= 5000000000
$AltVol=”/vol/Vol10”
$AltLun=”/vol/Vol10/LUN31”
$HostIgroupName=”TK2BBDSXX01”
$TagetVolObj=Get-NaVol $targetvol
If ( $LunSize –gt $TargetVolObj.SizeAvailable)
{     # conditional commands go here
      New-NaLun $TargetLUN $LunSize –type windows
      New-NaLunMap $TargetLUN $HostIGroupName
}
Else
{     New-NaVol $AltLUN
      New-NaLunMap $AltLUN $HostIGroupName
}
```

3.6 USE FOREACH LOOPS

Another method that is used to walk through a list of objects is to use a ForEach Loop, which lets you gain access to each element in each iteration of the loop. In this example, only the LUN paths to the individual LUNs are listed, and all the other information is skipped. This can be done either within a script or directly from the command prompt. A new variable that represents the current object in the collection of objects is created in the definition of the loop. In the following example, the current object is called `$LUN`.

```
PowerShell Prompt> $LUNList = Get-NALUN
PowerShell Prompt> ForEach ($LUN in LUNList) { $LUN.Path}
```

The following example uses commands to output to a screen with more human readable content and to add more information. LUNs that are not thin provisioned are also excluded.

```
Write-Host “The Following LUNs are Not Mapped”
$LUNList=Get-NaLun
ForEach ($LUN in $LUNList)
{     if ($LUN.Mapped)
            { # Since that returns a True/False, don’t need to compare }
      Else
            { Write-Host “LUN Name =”$LUN.path}
```

BREAK OUT OF A LOOP

Knowing how to break out of a loop is also helpful. You can use a **`For Each`** loop to walk through a large number of LUNs looking for something specific, but, once that specific thing is found, you want to break out of the loop and continue with the script. The **`Break`** command is designed for this purpose. The **`Break`** command is used to break out of, but not to end, the script. The following script determines if any unmapped LUNs exist in the array. If an unmapped LUN exists, the user should be informed, but the script does not need to continue.

```
For each ($LUN in Get-NaLun)
{     If ($LUN.Mapped –eq $false)
      {     Write-Host “Found an Unmapped LUN”
            Break;
      }
}
```

3.7 PLACE MANY COMMANDS INTO A SINGLE CLI LINE

To separate multiple commands that you want to execute, use the "**;**" character. This character enables you to run more complete commands from the CLI. For example, the commands in the previous console block can be completely run from a single cut and paste as follows:

```
PowerShell Prompt> $LUNList=Get-NaLun; ForEach ($Lun in $LUNList) { if
($LUN.Mapped) {# since that returns a True/False, don't need to compare} Else
{Write-Host "LUN Name ="$LUN.Path}
```

You can also short-circuit the requirement of creating a temporary variable and remove the comment. A built-in variable that equals false can also be used so that the IF statement does not need an Else clause.

```
PowerShell Prompt> ForEach ($LUN in get-NaLun) { if ($LUN.Mapped –eq $False)
{ Write-Host "LUN Name ="$LUN.Path } }
```

3.8 USE WILDCARDS

To make commands smarter, use wildcards as part of the command argument. For example, you can use a wildcard to get a list of LUNs that only includes LUNs that are part of a specific volume called /vol/DPM1.

- To only return the results that start with /vol/DPM1, execute the following command:

```
PowerShell Prompt> Get-NaLun /vol/DPM1/*
```

3.9 USE PIPELINE OUTPUT

Another method that allows very powerful but very short commands is piping the output of one command into the input of the next command. For example:

- To set the Aggregate called TestAggr offline and then remove the aggregate from the controller, execute the following command:

```
PowerShell Prompt> Set-NaAggr TestAggr -Offline | Remove-NaAggr
```

The following scenario depicts a more complex example of chaining commands. A LUN is made available to a Windows Cluster, and the Cluster uses the built-in Microsoft Multipathing IO (MPIO) instead of the enhanced MPIO Device Specific Module (DSM) that NetApp provides. Complete the following steps for this scenario:

1. Change the ALUA setting of all the Igroups that pertain to this cluster from ALUA = False to ALUA = True.
2. Use the name of one LUN, if known, that is currently mapped to the entire cluster to trace back the names of all of the Igroups in the cluster.
3. Feed those results into the command to set the ALUA setting for each Igroup to True.

```
PowerShell Prompt> Get-NaLunMap /vol/ClusterVol1/Q | Set-NaIgroup alua true
```

3.10 FILTER OUTPUT USING PIPES

You can also filter output to only display items that meet or exceed certain criteria. If you only want volumes to display that are larger than 100GB, then you can use a ForEach Loop with an If test in the middle to only output those LUNs.

1. To use a pipe to run the output from the first command into the next command, execute the following command:

```
PowerShell Prompt> Get-NaLun | where { $_.size –gt ‘1000000000’}
```

Note: A special pipeline variable, $, is used. This special variable acts as a placeholder for the current object.

2. Take the output of the second command, feed it to another filter, and only show LUNs that are currently mapped by executing the following command:

```
PowerShell Prompt> Get-NaLun | where { $_.size –gt ‘1000000000’} | where {
$_.mapped }
```

3. Find all of the LUNs that are on a controller and that are currently offline and bring them all online by executing the following command:

```
PowerShell Prompt> Get-NaLun | where {$_.online –eq $false } | set-NaLun –
online
```

3.11 BUILD FUNCTIONS WITHIN CODE

Often, building functions makes more sense than inserting common code in many places throughout your scripts. A function can be called with variables and it can return variables, or it can simply process a common set of commands.

The following example is a function that was created to accept an `Igroup Name` and a `LUN Name`, and to return a value of `True` if the Igroup is currently mapped to that LUN and to return a value of `False` if the mapped Igroup does not exist. Commands that fail send red error messages back to the users; these messages are not user friendly. Therefore, in this example, the requested script behavior is to process commands without ever sending red error messages. The script also assumes that the last controller that you connected to is the same controller that you want to run the command against.

```
Function CurrentlyMounted($IGroupName, $LunName)
{     $test=Get-NaLunMap $LunName
      If ($test –eq ‘’)
      {     # No maps exist for this LUN
            Return $false
      }
      Else
      {     $result=$false
            For each ($Map in $test)
            {     If ($Map.InitiatorGroupName –eq $IGroupName)
                  {     $Result=$true
                        Break
                  }
            }
            Return $Result
      }
}
```

The method to call this function might look like the one in the following example. This example shows two places in the code where either a LUN is mapped to a server or a LUN is unmapped from a server.

```
$ServerIG=”HYPV03”
$NewIG = “HYPV08”
$Lun=”/vol/vol7/Data17”
Write-Host “Attempting to Un-mount Server “$ServerIG” to LUN “$Lun
If (CurrentlyMounted($ServerIG,$Lun))
```

```
      {     Remove-NaLunMap $lun $ServerIG            }
Else  {     Write-Host “Lun was already dismounted”   }
Write-Host “Attempting to Mount Server “$NewIG” to LUN “$Lun
If (CurrentlyMounted($ServerIG,$Lun))
      {     Write-Host “Lun was already mounted”            }
Else  {     Add-NaLunMap $Lun $NewIG                        }
```

Note: Because the function in this case returns a value of either `True` or `False`, it can be used to control the `IF` statement by itself.

3.12 USE WHILE LOOPS TO WAIT FOR ACTIONS

A number of actions on a controller might take time to complete; for example, a NetApp SnapMirror® resync. In these situations, you want to wait until a command has completed before continuing with the script. However, this situation has two areas of concerns: You don’t want to wait longer than necessary by simply using a countdown timer, but you also want to avoid being stuck in an endless loop of waiting for an action that never completes.

The example script completes the following steps:

1. Make sure that the SnapMirror session is idle before starting.
2. Start the resync.
3. Wait until SnapMirror goes back to an `Idle` state before continuing (to make sure that a mirror is truly up to date before a **`Break`** command is issued to a SnapMirror).
4. Incorporate a timer to make sure that no more than 10 minutes elapse while waiting for the resync to return to `Idle` before error out.

```
Function CheckSnapMirrorBusy ($SnapName)
{     $StartTime=Get-Date
      $EndTime=$StartTime.AddMinutes(10) #setting timeout of 10 minutes
       $x1 = Get-NaSnapMirror $SnapName
       $x2=$x1.State
       If ($x2 –eq “broken-off”)
       {    Write-Host “SnapMirror Session Already Broken Off”
             Return $False
       }
       $x2=$x1.status
       While ($x2 –ne “idle”)
       {      $Timespan=new-timespan $StartTime $EndTime
              If ($Timespan –gt 0)
              {   Write-Host “Waiting until SnapMirror Session is Idle.”
                  Start-sleep(10)   #seconds
                  $x1=Get-SnapMirror $SnapName
                  $x2=$x1.status
              }
              Else
              {         Write-Host “Timer Exceeded”
                        Return $false
      }
      If ($x2 –eq “idle”)
      {     Write-Host “SnapMirror Session is Idle”
            Return $True
      }
}
```

This example script shows where the function is called in the main code block. This script completes the following steps:

1. Connect to the storage controller stored in the `$FromFiler` variable.
2. Check the SnapMirror status for the SnapMirror with the name stored in `$FromSnap`.
3. Wait for up to 10 minutes or until that SnapMirror is idle.
4. Continue if idle and issue a `Last Update` command.
5. Wait again until that SnapMirror session is idle.
6. Issue the **`Break`** command after it has returned to idle again.

```
…
ConnectFiler($FromFiler)#function that connects to the correct filer with
credentials
If (checksnapmirrorbusy($FromSnap))
{     Write-Host “About to Issue SnapMirror Update Command to SnapMirror”
      Invoke-NaSnapMirrorUpdate –source $FromSnap –destination $ToSnap
      If (CheckSnapMirrorBusy($FromSnap))
      {     Write-Host “About to break SnapMirror
            Invoke-NaSnapMirrorBreak –destination $ToSnap
}
…
```

The real value of this code is that it prevents certain types of behavior. An **`Update`** command is only issued if the session is idle, and a **`Break`** command is not issued after an **`Update`** command until the SnapMirror has again made the transition back to an idle state. This makes sure that the final updates have made it from site to site. This is an example of doing a controlled site-to-site failover.

3.13 USE STRING MANIPULATION

Often, a subset of a string needs to be gathered from a string. For example, if you have the path to a LUN such as /`vol/Volume37/DataLUN3`, you might want to run an operation on the volume. In this case, the volume name must be recovered to run an operation against the volume. Therefore, the following script separates just the volume name or the volume + qtree name.

The split function comes in very handy in this situation. Use the **`‘/’`** as the delimiter and the following commands to populate the variables:

```
$LUNList=get-nalun
For each($Lun in $LunList)
{     $Path=$Lun.Path
Write-Host “The Path to the LUN = “$Path
      $FirstSplit=$Path.Split(‘/’)[0]     # this will be a blank, since no
character exists before the first /
      $SecondSplit=$Path.Split(‘/’)[1]    # this will be the word ‘vol’
      $ThirdSplit=$Path.Split(‘/’)[2]     # this will be the volume name
      $FourthSplit=$Path.Split(‘/’)[3]    # this is either the QTree or the
LUN Name
      $FifthSplit=$Path.Split(‘/’)[4]     # if it exists then split3 was a
qtree, otherwise split3 was LUN
      Write-Host “Volume name = /vol/”$ThirdSplit
      If ($FifthSplit)
{     Write-Host “Qtree = “$FourthSplit” and LUN = “$FifthSplit
}
      Else
{     Write-Host “LUN =“$FifthSplit
```

```
}
}
```

Again, use the following two commands to find out what options exist to manipulate strings and research those separately:

```
PowerShell Prompt> $temp=”/vol”
PowerShell Prompt> $temp | Get-Member
```

Each of these methods manipulates strings in different ways; for example:

- **`ToUpper`** converts a string to all uppercase.
- **`Contains`** looks for a passed string within the search string.
- **`Replace`** can replace a set of characters with a new set or can even remove characters.

String manipulation can be used in endless ways, and this document only illustrates the possibilities. For a variety of examples on how to use each possible method, conduct a Web search for `PowerShell + method-name`.

4 HOW TO GATHER INFORMATION FROM WMI SOURCES

Many variables can be more readily obtained from the Microsoft Windows Management Instrumentation (WMI) source. Obtaining information from WMI is beneficial because it is the same source used for C# code, Visual Basic, Microsoft Windows Script Host (WSH), and a number of other languages. Microsoft has a tool available on the MSDN Web site that specifically identifies WMI data sources and how to access them. This tool, the Windows Scriptomatic 2.0, is available on:

- www.microsoft.com/downloads/en/details.aspx?familyid=09DFC342-648B-4119-B7EB-783B0F7D1178&displaylang=en

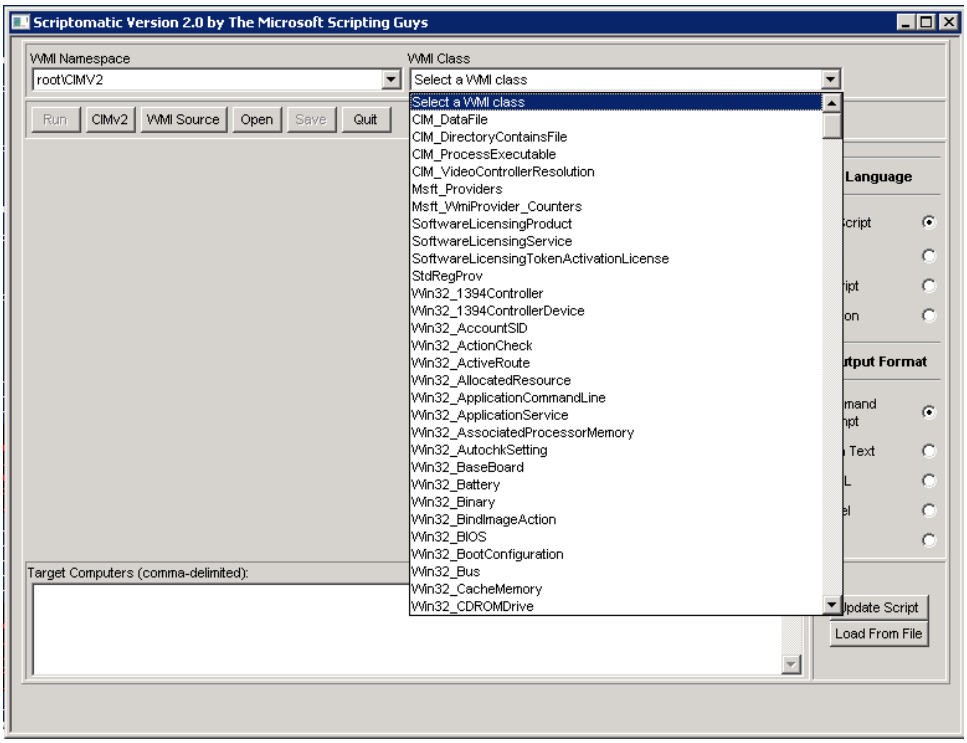
A specific version of PowerShell Scriptomatic is also available on:

- http://technet.microsoft.com/en-us/library/ff730935.aspx

Sometimes, information can be found faster by using the non-PowerShell-optimized Scriptomatic.

After you launch the Scriptomatic, sources for information display, as shown in Figure 5. The default WMI namespace is `root\CMIV2`, and most of the information that you need exists within this namespace. The WMI Class drop-down list contains most of the information that you need.

Figure 5) Scriptomatic – sources of information.

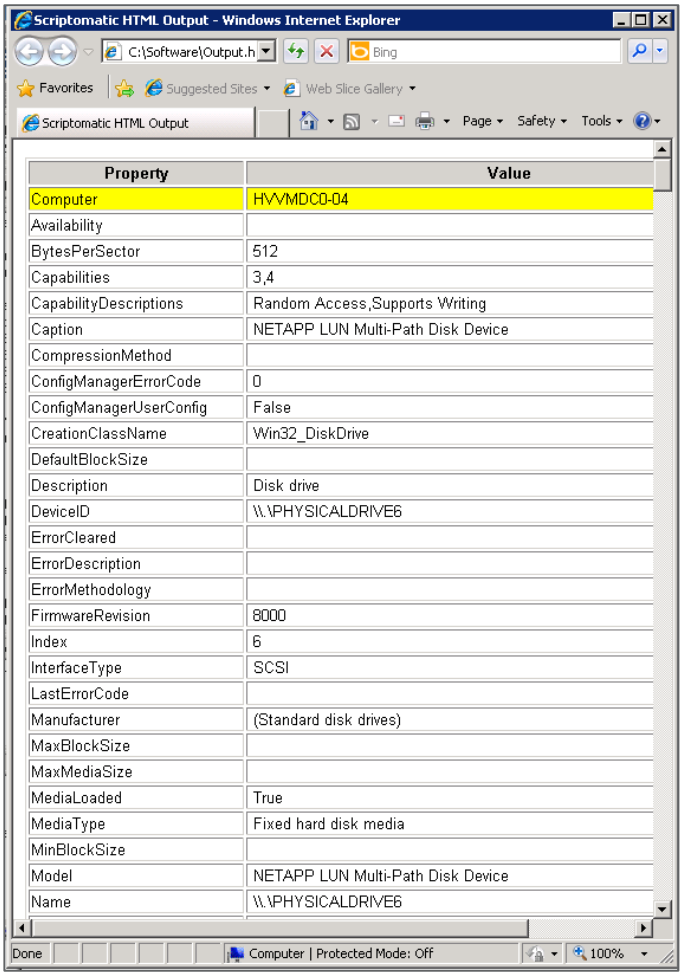


To obtain information from the Scriptomatic, complete the following example steps:

1. Select the WMI class `Win32_DiskDrive`.
2. Choose the HTML output format to make it readable.
3. Click Run the Script.

Figure 6 shows the output.

Figure 6) Scriptomatic HTML output.



As shown in Figure 6, the following output can be obtained:

- Retrieve fields from this WMI Class that are of interest.
- Read the name of the drive in a format that looks like `\\.\PHYSICALDRIVE6.`
- Read the model as a `NETAPP LUN Multi-Path Disk Device.`
- Scroll down to see the Serial Number, `P3h8W4Z-7Nwq`, and Size, `536946278400 bytes`, of the drive.

A variable to represent the collection of disk drives in the machine can be easily created using the following code. The code can then easily walk through the object showing the serial numbers of all of the detected drives.

```
$Drives = gwmi Win32_DiskDrive
ForEach ($Disk in $Drives)
{      $Diskname = $Disk.Name
       $DiskSerial = $Disk.SerialNumber
       Write-Host “Name of this Disk is “$DiskName” with Serial Number “$DiskSerial
}
```

Because you already know how to use the **`get-NaLun`** command to get information about a LUN, and you now know how to get information about a WinDisk, we can match them up. The following code uses two nested loops to find all of the LUNs that this server sees, and it returns the path that they exist at on the NetApp controller.

```
$Filer = “10.58.99.254”;
$admin= “administrator”
Connect-NaController $Filer –cred $admin
$LunList= Get-NaLun
$Drives = gwmi Win32_diskdrive
For each ($Disk in $Drives)
{      $DiskName = $Disk.Name
       $DiskSerial = $Disk.SerialNumber
       Write-Host “Name of this Disk is “$DiskName” with Serial Number “$DiskSerial
       ForEach ($Lun in $LunList)
       {    $LunPath=$Lun.Path
            $NASerial=Get-NaLunSerialNumber $LunPath
            If ($DiskSerial –eq NASerial){Write-Host “Exists on this path “$LunPath}
       }
}
```

5 HOW TO REDIRECT COMMANDS TO COMMAND-LINE PROGRAMS

Classes of programs exist that have information that is not available by using WMI or a PowerShell method. We can use redirected input and output to control these programs directly. Complete the following steps to take a predefined set of commands, execute Diskpart, and execute those commands.

1. Create a variable that contains the exact commands you want to execute, including the return characters; for example:

```
$PushCommand = @”
Select volume 8
Remove letter=s
Exit
“@
```

2. Send everything between the @” and “@ symbols to Diskpart by executing the following command:

```
$PushCommand | Diskpart
```

3. Select the following variables in the pushed command:

```
$Vols=@(“1”,”2”,”7”,”8”,”9”,”10”)
For each ($volnum in $Vols)
{
$pushcommand = @”
Select volume”$volnum”
Remove letter=s
Exit
“@
$pushcommand | diskpart
}
```

4. Run Diskpart six times in this command set.
5. Attempt to remove the drive letter `S` from a volume listed in the `$Vols` collection each time.
6. Capture the output from this command by assigning it to a variable; for example:

```
$outputdata = $PushCommand | Diskpart
```

In this case, the line-by-line responses to the commands that are pushed to Diskpart are fed into the variable named `$outputdata`.

7. Use string manipulation to parse for information.

5.1 OUTPUT RESULTS TO FILE INSTEAD OF SCREEN

To output the results to a file instead of to the screen, complete the following steps:

1. Consider defining the location and filename for the log file near the start of a script; for example:

```
$outfile = "C:\logs\script23.log"
```

2. Add lines to an out-file simply by creating a variable that you want logged.
3. Use the **`Pipe`** command directly to the PowerShell `command out-file`.

```
$outfile = "C:\logs\script23.log"
$logdata = "I want this line of text to appear in the Log File"
$logdata | out-file $outfile
Get-NaLun | out-file $outfile
```

6 SUMMARY

This guide shows how PowerShell is a valuable replacement for Telnet-type CLI functionality while also enabling greater amounts of automation by using scripts for complex commands. The major PowerShell benefits as a Telnet replacement are its easy learning curve and immediate functionality without learning any of the complexities of the language. If you want to use more advanced functions (such as loops), then the CLI commands can execute operations on a target NetApp device, which could not be done from a Telnet CLI.

The Data ONTAP PowerShell module is updated constantly. Therefore, check the NetApp Communities Web site so that you have the most recent version. Each updated version adds new features and new capabilities. Also, check for submitted scripts that can accomplish commonly used administrative tasks to avoid reinventing the process each time. NetApp also recommends posting scripts of tasks that you commonly run into in your arrays on the community site so that others can benefit from your scripts of CLI commandlets.

7 APPENDIX

7.1 COMMON TOOLS

A number of common tools can be used when writing PowerShell scripts for an environment. A few of these tools are covered in the body of this document, but many are not. This appendix covers some tools that increase productivity. These tools can all be downloaded free from the Web.

NOTEPAD++

Notepad++ is preferable to the basic notepad that is included with Windows because it does a fairly good job of indenting to keep code readable. In addition, it understands the language enough to help isolate problems such as an incorrect number of open-to-close brackets or misspelled commands. This tool can be found at http://notepad-plus-plus.org/.

If you download and use this tool, make sure to set the language to PowerShell because it is language aware. This tool is not as complete as the PowerShell Integrated Scripting Environment (ISE), but it is very lightweight and approachable.

CODEPLEX

CodePlex is more of a Web site with a collection of valuable tools. The Web site hosts a number of PowerShell modules that can be downloaded. For example, you can download the Hyper-V PowerShell Module that allows complete control of a Hyper-V virtual environment from your PowerShell scripts. More modules as well as sample scripts are also available on this Web site. Click www.codeplex.com and search for "PowerShell" to get started on this Web site.

MICROSOFT POWERSHELL TEAM BLOG

The Microsoft PowerShell team writes this blog. This blog is a good place to find out what's being done by your peers and to keep current on new and future developments in PowerShell. Click http://blogs.msdn.com/b/powershell/ to read this blog.

NETAPP POWERSHELL

The NetApp PowerShell Community Web site contains much more detailed information that relates specifically to how to implement NetApp features using PowerShell. This site is frequented by the authors of the NetApp PowerShell kit and by PowerShell experts who are either NetApp employees or NetApp customers:
http://communities.netapp.com/community/interfaces_and_tools/data_ontap_powershell_toolkit.

8 REFERENCES

The following Web sites are referenced in this document and in the appendix:

- CodePlex Open Source Project Community: www.CodePlex.com
- Microsoft Download Center Scriptomatic 2.0: www.microsoft.com/downloads/en/details.aspx?familyid=09DFC342-648B-4119-B7EB-783B0F7D1178&displaylang=en
- Microsoft PowerShell Scriptomatic: http://technet.microsoft.com/en-us/library/ff730935.aspx
- Microsoft Script Center: http://technet.microsoft.com/scriptcenter/dd742419.aspx
- NetApp Community; Developer Tools; Data ONTAP PowerShell Toolkit: http://communities.netapp.com/community/interfaces_and_tools/data_ontap_powershell_toolkit
- NetApp for Microsoft Environments Blog: http://blogs.netapp.com/msenviro
- NetApp on the Web: http://now.netapp.com
- Notepad++: http://notepad-plus-plus.org/
- Windows PowerShell Blog: http://blogs.msdn.com/b/powershell/

9 ACKNOWLEDGEMENTS

The contributions from the following people have greatly enhanced the manageability of NetApp devices by using PowerShell.

- **Jeffery Snover.** Microsoft Distinguished Engineer who developed the PowerShell concept and language.
- **Clinton Knight.** Author of the NetApp PowerShell Toolkit. He diligently updates the kit each quarter to both add functionality and design data formatters for more uniform output.
- **The NetApp MS Enviro Team.** The team members provided a peer review of this report and tested the applications. For more information about the MS Enviro team, go to the team's blog at http://blogs.netapp.com/msenviro/.

NetApp provides no representations or warranties regarding the accuracy, reliability or serviceability of any information or recommendations provided in this publication, or with respect to any results that may be obtained by the use of the information or observance of any recommendations provided herein. The information in this document is distributed AS IS, and the use of this information or the implementation of any recommendations or techniques herein is a customer's responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. This document and the information contained herein may be used solely in connection with the NetApp products discussed in this document.

www.netapp.com