Windows PowerShell 2.0 Best Practices Quick Reference Guide

Windows PowerShell 2.0 is so powerful and allows for so many different ways to perform the same task that, in some cases, this flexibility can become confusing. Throughout this book numerous best practices are identified, and they are collected here for your convenience. For more information about these best practices, refer to the chapter in which they are found.

Chapter 1 Best Practices

- Set the *#requires version 2.0* tag in any script that uses a feature from Windows PowerShell 2.0.
- Use the `Get-EventLog` cmdlet when accessing the classic event logs. Use `Get-Event` when working with the newer diagnostic and tracing logs.
- Use a remote interactive session when you wish to perform multiple commands on a single remote computer. If you have one or two commands you wish to run on multiple remote computers, use remote command execution. If you have multiple commands you wish to execute on multiple computers, use a persistent connection. If you need to run a script that does not exist on a remote computer, use remote script execution.
- Be very careful when using the inline method of working with COM objects. Inline code, in which you both create and use the object at the same time, can quickly become difficult to read. Stylistically, you should prefer the block method, in which you create the object and store it in a variable. Then, in the next line, you use the methods or properties from that object.
- Prior to deploying Windows PowerShell 2.0, you will need to decide whether you will take advantage of the remoting features, and whether you will utilize the graphical features of PowerShell. If the answer is no on both counts, then you can skip some of the deployment steps.
- When using the *Measure-Command,* you should not make decisions about the capabilities of commands based upon milliseconds. Rewrite your task in the form of a script that performs several thousand iterations of the command and time that.

- Use the native PowerShell cmdlets instead of the corresponding COM objects because they are easier to use. The only exception to this rule is when a COM object has a clear speed advantage over the PowerShell cmdlet.
- Initially deploy PowerShell to servers. It is likely that this is where you will see the greatest benefit. Once ensconced in the server farm, you should deploy to the desktop. Training should follow deployment—server administrators, then help desk personnel.

Chapter 2 Best Practices

- If you need to modify a property on a WMI object, and the command needs to be done in a single line, use the `Set-WMiInstance` cmdlet.

Chapter 3 Best Practices

- Always include a help function when defining command line parameters for a script.
- Use a very/noun form for function names.
- When possible, stay with standard verbs when choosing verbs for function names.
- Use Microsoft Excel to simplify spacing and quoting issues when creating Comma Separated Value (CSV) files.

Chapter 4 Best Practices

- Use command-line parameters to allow for the modification of the behavior of the script at run time when a script will be used as a utility script.
- When using a switched parameter, always check for the presence of the switch and include a default behavior for the parameter.
- When calling a cmdlet from inside of a script, always use the full parameter name. Do not rely upon either positional parameters or partial parameter completion.

Chapter 5 Best Practices

- When using the *StdRegProv* WMI class, call the class from the root\default WMI namespace, not from the root\cimv2 namespace. This will ensure backward compatibility with earlier versions of the Windows operating system.
- A script should detect the rights the script has when running and compare it with any requirements the script may need to run properly.

Chapter 6 Best Practices

- Prior to creating a new alias, see if there is a suitable alias already created for the cmdlet in question.
- When creating a new alias, use either the `New-Alias` or the `Set-Alias` cmdlet.
- You should not create constant aliases unless you are certain you do not need to modify or delete them.
- When naming functions, use the verb/noun naming convention because this syntax will be familiar to users of Windows PowerShell, and because you can take advantage of tab expansion.
- Include a comment that indicates the bottom curly bracket closes the function.
- Always apply type constraints to your function parameters.
- When naming parameters, consider how partial parameter completion will work with the parameter name.
- When building up file paths, use the path cmdlets, such as `Join-Path`, to avoid concatenation errors.
- Specify the description attribute when creating a PowerShell drive.
- When creating standard aliases and variables, mark them as constant to ensure they will always be available.
- When creating functions and PowerShell drives, choose names that avoid potential naming conflicts.
- Always use the `Test-Path` cmdlet to verify a file's existence whenever you use an include file in your script.

Chapter 7 Best Practices

- Check the version of the operating system prior to executing the script if you know that there could be version compatibility issues.
- Line up the curly brackets from a script block unless the command is very short and will fit easily onto a single line.

Chapter 8 Best Practices

- Pipeline the results of the fan out command to a `Format-Table` cmdlet and choose the *machineName* property.
- Always choose the property with the longest values to be displayed for the last position in the command when formatting output as a table.

- Use the `Get-PsSession` cmdlet to obtain a listing of the all the PsSessions on the computer.
- Remove disconnected sessions when you do not think you will be going back to the target computer in the near future.

Chapter 9 Best Practices

- Rather than display a raw error message, which most users and many IT Pros find confusing, suppress the display of the error message and perhaps inform the user an error condition has occurred, and provide more meaningful and direct information that the user can then relay to the help desk.
- When using more than two input parameters, modify the way the function is structured to make the function easier to read.
- Avoid creating functions that have a large number of input parameters.
- A function should be narrowly defined and should encapsulate a single thought.

Chapter 10 Best Practices

- Place the multiline comment tags on their own individual lines.
- Use comments to explain new or novel script structures.

Chapter 11 Best Practices

- Supply the *.synopsis* and the *.example* tag due as this is the most critical information required to assist a person in learning how to use a function.
- When using wildcards, load modules to type a significant portion of the module name so that you match only a single module from the list of modules that are available to you.

Chapter 12 Best Practices

- Avoid using the *throw* statement unless an action actually causes an error.
- Use the parameter validation attributes to inspect parameter values rather than writing your own functions to accomplish the same thing.
- When working with connection strings to data sources, use variables to hold each portion of the connection string.
- Avoid using the `Out-Host` cmdlet or the `Out-Default` cmdlet unless there is a reason for using it.

Chapter 13 Best Practices

- Assign the default value for a parameter in the *param* statement.
- Do not make permanent modifications to the Windows PowerShell environment in a script.

Chapter 14 Best Practices

- If you find that a script runs only on Windows PowerShell 2.0, include the *#requires –version 2.0* tag in your script.
- Due to the asynchronous nature of the pipeline and the reduced memory footprint of the operation, engage the pipeline whenever it makes sense in your code.
- Use a *Write-Debug* statement before calling a method, value assignment, or a function.
- Implement the *whatif* parameter whenever the script does anything that changes system state.

Chapter 15 Best Practices

- If you want to modify the script execution policy on a local computer, use the `Set-ExecutionPolicy` cmdlet to make the change.
- If you have more than a few computers on which you need to set the script execution policy, use Group Policy to modify the Windows PowerShell script execution policy.
- Avoid direct editing of help desk scripts. You could inadvertently introduce errors or change the designed functionality of the script.

Chapter 16 Best Practices

- Hide errors from the user during the logon process because it avoids confusing the users and reduces help desk calls.
- Log the start time of the script as well as the end time of the script.
- Collect messages during script operation because a single I/O (input/output) operation can be undertaken to create the log file.
- The path to the special folders is cumbersome to manually derive; therefore, use the .NET Framework environment class to resolve the path.
- Any file that is very large, or that potentially could involve large amounts of data, should be created locally first and then copied to the network location.

- Because of the different caveats involved in working local files and folders, write to a temporary file in the temporary directory and then copy the temporary file to the networked share.

Chapter 17 Best Practices

- A Windows PowerShell script should be readable.
- A Windows PowerShell script should be understandable.
- While in a debugging session, selectively enable and disable breakpoints to see how the script is running while you are troubleshooting the script.