**10 Tips for the SQL Server PowerShell Scripter**

[The Scripting Guys](http://social.technet.microsoft.com/profile/The%20Scripting%20Guys)

FROM: http://blogs.technet.com/b/heyscriptingguy/archive/2013/05/06/10-tips-for-the-sql-server-powershell-scripter.aspx

The Scripting Guys

Microsoft Corporation

237,825 Points723

Recent Achievements

First Helpful VoteBlogs All-StarNew Gallery Contributor

[View Profile](https://social.technet.microsoft.com:443/profile/the%20scripting%20guys/?ws=usercard-hover)

6 May 2013 1:01 AM

* Comments[9](http://blogs.technet.com/b/heyscriptingguy/archive/2013/05/06/10-tips-for-the-sql-server-powershell-scripter.aspx#comments)
* Likes

**Summary**: Microsoft PowerShell MVP, Chad Miller shares his top ten tips for the SQL Server Windows PowerShell scripter.

Microsoft Scripting Guy, Ed Wilson, is here. Today’s blog is a bit unique. I was throwing around ideas with Chad Miller, and somehow we came up with the idea that he would share some tips for the SQL Server Windows PowerShell scripter. You can read more about Chad and see his other blog posts on the [Hey, Scripting Guy! Blog site](http://blogs.technet.com/b/heyscriptingguy/archive/tags/chad+miller/).

And now, Chad…

**1. Use here-strings**

Here-strings are great for working with strings that span multiple lines or contain characters you would normally need to escape, such as embedded quotes. Like regular strings, when using here-strings, you still get variables replaced by their values.

$group = "PRD"

#Don't do this:

$query = "SELECT DISTINCT"

$query +="`n s.server\_name"

$query += "`n FROM msdb.dbo.sysmanagement\_shared\_registered\_servers s"

$query +="`n JOIN msdb.dbo.sysmanagement\_shared\_server\_groups g ON s.server\_group\_id = g.server\_group\_id"

$query += "`n WHERE g.name = '$group'"

#Instead use a here-string:

$query = @"

SELECT DISTINCT

s.server\_name

FROM msdb.dbo.sysmanagement\_shared\_registered\_servers s

JOIN msdb.dbo.sysmanagement\_shared\_server\_groups g ON s.server\_group\_id = g.server\_group\_id

WHERE g.name = '$group'

"@

Not only is the here-string example more readable, but you can also copy and paste it right into SQL Server Management Studio. You test your queries before running them in Windows Powershell, right? for For more information about here-strings, see [about\_Quoting\_Rules](http://technet.microsoft.com/en-us/library/hh847740.aspx).

**2. Leverage Central Management Server**

Some administrators may store their list of servers in text files or Active Directory organizational units, but you have a SQL Server Central Management Server (CMS). Use your CMS to pull lists of servers and SQL Server instances for input into Windows PowerShell scripts.

$serverInstances = sqlcmd -S myCMServerInstance -d msdb -Q $query -h -1 -W

$serverInstances | foreach {new-object Microsoft.SqlServer.Management.Smo.Server($($\_.server\_name)) } |

Select Name, PhysicalMemory, @{n='MaxServerMemory'; e={$\_.Configuration.MaxServerMemory.RunValue}}

#Get server names by removing instance name i.e. Z001\SQL1 becomes Z001

$servers = sqlcmd -S myCMServerInstance -d msdb -Q $query -h -1 -W | foreach { $\_ -replace "\\.\*|,.\*" }

invoke-command -ComputerName $servers -ScriptBlock {get-psdrive -PSProvider FileSystem}

For more information, see [Administer Multiple Servers Using Central Management Servers](http://msdn.microsoft.com/en-us/library/bb895144.aspx).

**3. Load SMO assemblies**

SQL Server Management Objects (SMO) is the API you'll use to script against SQL Server. To use SMO, you must first load the SMO assemblies. If you have SQL Server 2012, all you need to do to load assemblies is call **Import-Module sqlps**. This will load the SMO assemblies with the sqlps module. If you're not using 2012 or would prefer to load the assemblies without the sqlps module, you'll need to use **Add-Type** with the fully qualified name:

#For SQL Server 2008 R2 and SQL Server 2008

add-type -AssemblyName "Microsoft.SqlServer.ConnectionInfo, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.Smo, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.SMOExtended, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.SqlEnum, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.Management.Sdk.Sfc, Version=10.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

#For SQL Server 2012

add-type -AssemblyName "Microsoft.SqlServer.ConnectionInfo, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.Smo, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.SMOExtended, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.SqlEnum, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

add-type -AssemblyName "Microsoft.SqlServer.Management.Sdk.Sfc, Version=11.0.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" -ErrorAction Stop

#See loaded SMO assemblies

[appdomain]::CurrentDomain.GetAssemblies() | where {$\_.FullName -like "\*sqlserver\*"}

Like SQL Server Management Studio, SMO is backwards compatible so you can script against 2000 and 2005 SQL Server using either the 2008 or 2012 version of SMO.

**4. Exercise Invoke-Sqlcmd with caution**

One of the cmdlets that is included with SQL Server 2012, SQL Server 2008 R2, and SQL Server 2008 is **Invoke-Sqlcmd**. As the name implies, the cmdlet tries to be a Windows PowerShell version of the venerable sqlcmd utility introduced in SQL Server 2005 and enhanced with each SQL Server release. The **Invoke-Sqlcmd** cmdlet has [several bugs and workarounds that you should be aware of](https://connect.microsoft.com/SQLServer/SearchResults.aspx?SearchQuery=invoke-sqlcmd).

One issue is with the **QueryTimeout** setting. At the time of this writing, the documentation incorrectly states the following about **QueryTimeout** parameter:

help Invoke-Sqlcmd -Parameter QueryTimeout

QueryTimeout

Specifies the number of seconds before the queries time out. If a timeout value is not specified, the queries do not time out. The timeout must be an integer between 1 and 65535.

Required? false

Position? named

Default value

Accept pipeline input? false

Accept wildcard characters? false

This isn't true, which can be easily proven by running this simple test:

invoke-sqlcmd "waitfor delay '00:00:31'" -Database master -ServerInstance $env:computername

Invoke-Sqlcmd : Timeout expired. The timeout period elapsed prior to completion of the operation or the server is not responding.

So, you'll need to specify a query timeout. In SQL Server 2012, the bug was partially fixed and you can specify 0, which means no **QueryTimeout**:

invoke-sqlcmd "waitfor delay '00:00:31'" -Database master -ServerInstance $env:computername\sql1 -querytimeout 0

When you use the SQL Server 2008 R2 or SQL Server 2008 versions of **Invoke-Sqlcmd**, specifying 0 doesn't work. Instead, you'll need to provide a value greater than 1. Here's a trick that works:

invoke-sqlcmd "waitfor delay '00:00:31'" -Database master -ServerInstance $env:computername\sql1 -querytimeout ([int]::MaxValue)

One other issue that isn't a bug, but something to be aware of…

If you're querying data with columns larger than 4,000 characters for XML or char data types, or 1,024 bytes for binary data types, you'll want to override the default settings for **MaxCharLength** or **MaxBinaryLength** parameters:

#By default character data larger than 4,000 is truncated

invoke-sqlcmd "select REPLICATE('x', 4001)" -Database master -ServerInstance $env:computername\sql1 | foreach {($\_.Column1).length}

4000

#This is fixed by specifying a value for the maxcharlength parameter

invoke-sqlcmd "select REPLICATE('x', 4001)" -Database master -ServerInstance $env:computername\sql1 -maxcharlength ([int]::MaxValue) | foreach {($\_.Column1).length}

4001

#See help for more information on max length parameters

help invoke-sqlcmd -Parameter max\*

-MaxCharLength

Specifies the maximum number of characters returned for columns with character or Unicode data types, such as

char, nchar, varchar, and nvarchar. The default is 4,000 characters.

Required? false

Position? named

Default value

Accept pipeline input? false

Accept wildcard characters? false

-MaxBinaryLength

Specifies the maximum number of bytes returned for columns with binary string data types, such as binary and

varbinary. The default is 1,024 bytes.

Required? false

Position? named

Default value

Accept pipeline input? false

Accept wildcard characters? false

One serious issue with **Invoke-Sqlcmd** is around error handling. As an example, the following command does not produce an error in **Invoke-Sqlcmd** at the time of this writing:

invoke-sqlcmd -ServerInstance $env:computername\sql1 -Database tempdb -Query "select 1/0" -OutputSqlErrors $true -AbortOnError -SeverityLevel 0 -ErrorLevel 0

Sqlcmd.exe and SQL Server Management Studio return an error:

sqlcmd.exe -S "%COMPUTERNAME%" -d tempdb -Q "select 1/0"

Msg 8134, Level 16, State 1, Server YourServer, Line 1 Divide by zero error encountered.

Unfortunately, there aren't any good workarounds for the error handling issue other than don't use **Invoke-Sqlcmd** if you need error handling. There are other issues with **Invoke-Sqlcmd**, which the command-line utility, sqlcmd.exe, doesn't have.

In any case it's trivial to write your own **Invoke-Sqlcmd** replacement. For more information, see [Invoke-Sqlcmd2](http://gallery.technet.microsoft.com/scriptcenter/7985b7ef-ed89-4dfd-b02a-433cc4e30894) in the Script Center Repository.

You may want to continue to use sqlcmd.exe for scripted data loading.

**5. Don't forget SQL Server command-line utilities**

One of tenets of Windows PowerShell is to not needlessly write scripts. What I mean by this is instead of writing a script to perform an action, first see if there's a cmdlet that accomplishes the task. Think of a cmdlet as a script that you don't have to write.

And if there isn't a cmdlet available? Here's where Windows PowerShell (being a shell) comes in handy. You can still use the native Windows console applications. You might even find that if there is Windows PowerShell cmdlet, the console application simply works better. At the top of my list of SQL Server command-line utilities that are still very useful: **sqlcmd**, **dtutil**, and **dtexec**, and Red Gate **sqlcompare**.

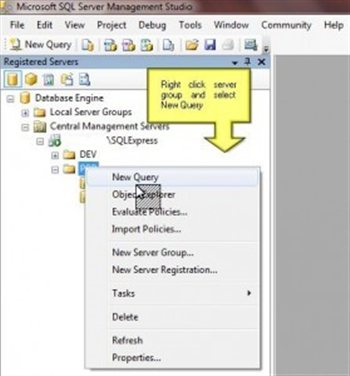
**6. Read the SMO documentation**

If you're going to write scripts that use SMO, you'll probably want to spend some time looking at the SMO documentation. Rather than hunt through the documentation, I find it easier to search the web for the SMO documentation on a particular class. Add the term "SMO class" to your web search, for example, "smo server class." The first item in your search results will usually be the MSDN documentation.

**7. Don't use Windows PowerShell for everything**

If you know T-SQL or SQL Server Reporting Services, the solutions you build can leverage the best tool for the job. What's really cool is using Windows PowerShell as only one part of the solution. One of my favorite patterns is to use Windows PowerShell to collect information, load the data into a SQL Server table, and then present the data as a web-based report in SQL Server Reporting Services.

Check out my previous post, [Use PowerShell to Collect Server Data and Write to SQL](http://blogs.technet.com/b/heyscriptingguy/archive/2010/11/01/use-powershell-to-collect-server-data-and-write-to-sql.aspx) for more information. Let's face it, sometimes using Windows PowerShell is more complex than what's needed. Do you need to interactively run query across multiple SQL Servers? It doesn't get much simpler than multiserver queries in SQL Server Management Studio.

[](http://blogs.technet.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-76-18/5367.hsg_2D00_5_2D00_6_2D00_13_2D00_1.jpg)

**8. Load data with Out-GridView**

Although you could mess with scripts to export data directly into Excel or convert CSV files, a quick and dirty way to get data into Excel is to simply copy and paste from **Out-GridView**. This also works on small tables you open in SQL Server Management Studio Object Explorer: right-click **Table** and select **Edit Top 200 Rows**.

You'll then be able to paste **Out-GridView** output rows into a SQL Server table. This is especially useful when you need to run a Windows PowerShell command one-time and insert the data into Excel. Here's an example script called [Get-SqlSpn](http://gallery.technet.microsoft.com/scriptcenter/Get-SqlSpn-17d5c0d7), which grabs all the SPNs in a domain related to SQL Server. I'll then use **Out-GridView** with CTRL+A and CTRL+C to copy, and then CTRL+V to paste into Excel.

. ./get-sqlspn.ps1

$spns = Get-SqlSpn

#4. Get Column Headers

$object = $spns | select -first 1

$ht = @{}

foreach($property in $object.PsObject.get\_properties()) {

$ht.add($property.Name.ToString(),$property.Name.ToString())

}

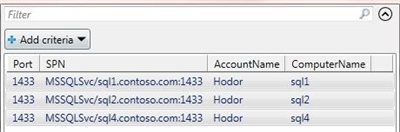
new-object psobject -Property $ht | out-gridview

#Copy/Paste heading row to Excel (Ctrl-A, Ctrl-C)

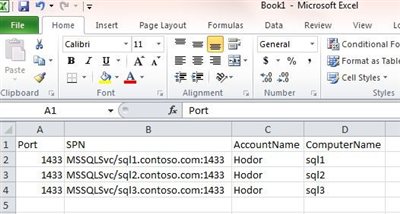
$spns | out-gridview

#Copy/Paste spns to Excel (Ctrl-A, Ctrl-C)

This image shows an example of where I copy from:

[](http://blogs.technet.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-76-18/8284.hsg_2D00_5_2D00_6_2D00_13_2D00_2.jpg)

And here is my paste into Excel:

[](http://blogs.technet.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-76-18/4555.hsg_2D00_5_2D00_6_2D00_13_2D00_3.jpg)

**9. Adjust SMO StatementTimeout**

The default statement time setting for SMO is 10 minutes. Of course, some operations (such as a backup or restore) can take longer than the default timeout of 600 seconds (10 minutes). Set the **StatementTimeout** setting to no timeout by specifying 0:

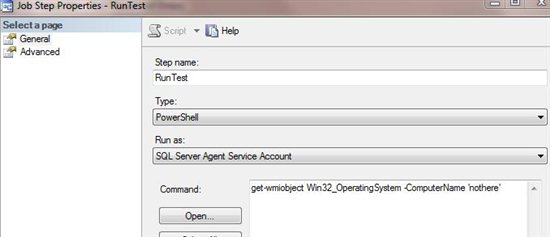
$server = new-object ("Microsoft.SqlServer.Management.Smo.Server") "Z001\SQL"

$server.ConnectionContext.StatementTimeout = 0

**10. Handle Windows Powershell errors in SQL Server Agent jobs**

By default the **ErrorActionPreference** is set to Continue, and this has implications on how errors bubble up to the SQL Server Job Server. If you run a Windows PowerShell command as a SQL Server Agent job and there are no syntax errors yet, the command produces an error (for example, attempting to get operating system information from an unavailable server). The SQL Server Agent job will report success. If you want an error condition to halt execution of a SQL Server Agent job or to produce an error, you'll need to add some error handling. You can set up a SQL Server Agent job with Windows PowerShell Job Step as follows:

get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere'

[](http://blogs.technet.com/cfs-file.ashx/__key/communityserver-blogs-components-weblogfiles/00-00-00-76-18/3441.hsg_2D00_5_2D00_6_2D00_13_2D00_4.jpg)

The job will run successfully, but if you run it directly in Windows PowerShell, you'll see:

get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere'

get-wmiobject : The RPC server is unavailable. (Exception from HRESULT: 0x800706BA)

At line:1 char:1

+ get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere'

To bubble up Windows PowerShell errors to SQL Server Agent, you'll need to do one of the following:

**A. Set your $ErrorActionPreference = "Stop"**

$erroractionpreference = "Stop"

get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere'

**B. Set ErrorAction at the cmdlet-level (more granular)**

get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere' -ErrorAction 'Stop'

**C. Use Try/Catch with ErrorActionPreference or ErrorAction**

try {

get-wmiobject Win32\_OperatingSystem -ComputerName 'nothere' -ErrorAction 'Stop'

}

catch {

throw "Something went wrong"

#or rethrow error

#throw $\_

#or throw an error no message

#throw

}

**D. Continue, and fail the SQL Server Agent job**

Let's say you have a collection of computers and you want to continue on the error, but you also want to fail the job. In this case you can make use of the **ErrorVariable**:

#Note the -ErrorVariable parameter takes a variable name without the $ prefix.

get-wmiobject Win32\_OperatingSystem -ComputerName 'localhost','nothere','Win7boot' -ErrorVariable myError

if ($myError)

{ throw ("$myError") }

~Chad

Thank you, Chad, for some extremely practical information. Well done! Join me tomorrow for the exciting conclusion to our SQL Server Week as we have a guest blog written by SQL Server MVP, Kendal Van \*\*\*.

I invite you to follow me on [Twitter](http://bit.ly/scriptingguystwitter) and [Facebook](http://bit.ly/scriptingguysfacebook). If you have any questions, send email to me at [scripter@microsoft.com](mailto:scripter@microsoft.com), or post your questions on the [Official Scripting Guys Forum](http://bit.ly/scriptingforum). See you tomorrow. Until then, peace.

**Ed Wilson, Microsoft Scripting Guy**

* [9 Comments](http://blogs.technet.com/b/heyscriptingguy/archive/2013/05/06/10-tips-for-the-sql-server-powershell-scripter.aspx#comments)

[Comments](http://blogs.technet.com/b/heyscriptingguy/rsscomments.aspx?WeblogPostID=3570404)

* Mbourgon

7 May 2013 5:07 AM

Awesome tips. Thanks, Chad. I actually took your article on and used it to build an extensible repository - drop scripts in a folder, it runs them against all your machines in parallel, (several at a time) and saves to a table. Invoke-sqlcmd2, out-datatable and all. (Now to figure out why it's not parsing the string sizes correctly \*grin\*). Looking forward to trying several of these other tricks.

Now if we could only get Mr. Van \*\*\*'s name posted correctly. :)

* Phil Factor[Phil Factor](http://social.technet.microsoft.com/profile/Phil%20Factor)

Phil Factor

5 Points200

Recent Achievements

New Blog CommentatorFirst Forums Reply

[View Profile](https://social.technet.microsoft.com:443/profile/phil%20factor/?ws=usercard-hover)

14 Oct 2013 4:09 PM

Some great advice here, Chad. I agree that it is generally best to use the SQLCMD utility itself rather than the rather quirky invoke-sqlcmd, especially when using the colon commands, but I've always found it tricky to import the text from SQLCMD into a form that is easy to convert to PowerShell objects so I'd appreciate any advice you have about a slick way of doing this.

I must ask Kendal Van \*\*\* if he has any tips!

* Joel Newton

27 Feb 2014 6:18 PM

Hi, Ed,  
Thanks for the article. Are there recommended ways for managing SQL client connections in PS? I have an environment with about 600+ Windows Servers who all connect via PS to a SQL db for configuration settings and logging. I use the System.Data.SQLClient.SQLConnection object for making connections and close when done, but at any given time I'm seeing 150 sleeping connections from my servers on the database. Hence I need to override the default max connection limit of 100. Should I be doing something other than / in addition to calling $connection.close()?  
  
Thanks,  
Joel

* Joel Newton

27 Feb 2014 6:35 PM

I just found an answer to my question. Since the majority of connections are sleeping, I'm most likely closing them out correctly, but since I could have many more than the default max of 100 at a given time, I need to specify a higher max pool size in all my connection strings.

* Adam D

19 Mar 2014 3:49 PM

cheers mate

* Thomas

10 Jun 2014 10:29 AM

Nice article, I'm new to using PowerShell to handle SQL relative tasks, it does help me on the concept.