Practical PowerShell for SQL Server Developers and DBAs – Part 2

15 August 2012

by Michael Sorens

Having shown just how useful PowerShell can be for DBAs in executing queries, Michael Sorens now takes us through navigating SQL Server space and finding meta-information - valuable information for anyone looking to be more productive in SQL Server.

Practical PowerShell for SQL Server Developers and DBAs – Part 1

▶ Practical PowerShell for SQL Server Developers and DBAs – Part 2

Contents

If you have not done so already, please review Part 1 to get the most out of this article. Also see the accompanying wallchart (link starts PDF download) that distills the key details out of both parts into a one-page reference.

Navigating in SQL Server Space

Data Stores and Drives

If all you could do was execute queries, PowerShell would be a weak cousin indeed to something like SQL Server Management Studio. But PowerShell allows you to navigate and explore "SQL Server space" from the command line in a similar way that you would use SSMS's object explorer and context menus. Before tackling this capability, however, you need to have an appreciation for PowerShell *data stores*. First, realize that these *data stores* are quite distinct from SQL Server *databases*. Each PowerShell data store is managed by a *PowerShell provider* (see the PowerShell help page about_providers for more details). If you run the **Get-PSProvider** cmdlet you will see a list similar to this:

Name	Capabilities	Drives
WSMan	Credentials	{WSMan}

Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, D, E, F}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}
SqlServer	Credentials	{SQLSERVER, DB}

Start with the most familiar provider, the **FileSystem provider**. On my system it manages four drives, the ubiquitous **C:** drive along with the **D:**, **E:**, and **F:** drives. As you are likely well aware, you can set your current location to any of these drives and then navigate around the drive to access various programs and files. Unlike the old Windows command shell, you do both of these commands (going to a drive or going to a folder) exactly the same in PowerShell:

```
Set-Location C:
Set-Location Documents
Set-Location C:\usr\tmp
chdir C:\usr\tmp # alias for Set-Location
cd C:\usr\tmp # alias for Set-Location
sl C:\usr\tmp # alias for Set-Location
```

Once you go to a particular location, you can list the contents using the **Get-ChildItem** cmdlet (aliases: dir, gci, ls). Or, as you might expect, you could specify a path to **Get-ChildItem** without changing your current location (e.g. `ls C:\usr\tmp`).

PowerShell has abstracted this concept in a novel and powerful way to other data stores. Notice, for example, that there is a *Registry provider*, which by default provides two "drives", the **HKLM:** and **HKCU:** drives. Just as you would expect, you can set your location to either of these drives, then navigate the registry tree with the same **Set-Location** cmdlet. Filesystems and registry hives have a hierarchical structure; other drives managed by the providers shown do not. For example, you can navigate to the environment data store (`cd env:`) but there is nowhere else to go from there. What you can do, however, is list the contents of the drive (`ls env:`).

Now take a look at the *SqlServer provider*, managing two drives on my system. The first (SQLSERVER:) is the default you will find whenever you have the SQL Server provider installed. The second is a custom drive that I have instantiated; more on that later. This SQL Server provider is courtesy of the familiar **sqlps** module, discussed earlier. When you import the module not only does it load SQL Server cmdlets but it also loads the SQL Server provider and auto-mounts the SQLSERVER: drive.

Nodes in SQL Server Space

The SQLSERVER: drive provides a number of root-level items that you can explore. The main one of interest (discussed below) is \SQL which contains all your database objects; others contain policy, server, integration service, and analysis service objects—see the root-level hierarchy detailed on the SQL Server PowerShell provider page on MSDN for more details on these other object types.

The table below specifies the paths to some of the locations you will most commonly use.

Description	Node
SQL Server data store root	SQLSERVER:\
Network root	SQLSERVER:\SQL

Instances on selected machine	SQLSERVER:\SQL*machine*
Top-level objects in selected instance	SQLSERVER:\SQL*machine**instance*
Databases in selected instance	SQLSERVER:\SQL*machine**instance*\Databases
Top-level objects in selected database	SQLSERVER:\SQL*machine**instance*\Databases*database*
Tables in selected database	SQLSERVER:\SQL*machine**instance*\Databases*database*\Tables
Views in selected database	SQLSERVER:\SQL*machine**instance*\Databases*database*\Views
Roles in selected database	SQLSERVER:\SQL*machine**instance*\Databases*database*\Roles
Triggers in selected database	SQLSERVER:\SQL*machine**instance*\Databases*database*\Triggers

When navigating in *file system space*, you may either change your current location to a particular node (e.g. `Set-Location C:\a\b\c`) or from your current location you may reference a particular node (e.g. `Get-ChildItem C:\a\b\c`). In *SQL Server space* you have the same options. You can, for instance, `Set-Location SQLSERVER:\SQL\localhost` or you could **`Get-ChildItem`** `SQLSERVER:\SQL\server509\DEFAULT\Databases\sandbox\Tables`. As you might expect, the machine in a path may either be a specific server name or you can use **localhost** to refer to your current machine.

All the nodes under the \SQL node are deterministic; each time you display the contents of \SQL\localhost\SQLEXPRESS\ you get a list of all your databases. But the top-level \SQL node itself is unique. This displays a list of the machines you have touched in the current session. Assuming your machine name is `gandalf`, in a fresh PowerShell session `Get-ChildItem \SQL` will return two entries, `gandalf` and `localhost`. Each time you connect to a different server by naming it in a path (e.g. `Get-ChildItem SQLSERVER:\SQL\server509\DEFAULT`) you add that machine to the list of known servers so henceforth it will also appear when you list the contents of the top-level \SQL node. Thus, after the first such reference, `Get-ChildItem SQLSERVER:\SQL` returns two entries for your local machine (the actual name and `localhost`) plus any machines you have referenced, in this case `server509`:

```
Get-ChildItem sqlserver:\sql
MachineName
-----------
gandalf
localhost
server509
```

This example also illustrates a PowerShell feature that is not obvious (until you see it the first time): *the output of a cmdlet varies based on the provider*. In contrast to the output of the SQL Server provider on the \SQL node, for the file system provider **Get-ChildItem** returns output like this, showing properties you would expect for files and folders:

```
Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
d----          5/21/2012   9:58 AM            Modules
-a---          6/12/2012   6:44 PM        911
Microsoft.PowerShell_profile.ps1
-a---          6/18/2012   5:42 PM      33392 pshist.xml
```

Furthermore, **Get-ChildItem** could even return *different* output within a *single* provider

based on context. The table below lists the same common SQL Server paths shown earlier, this time listing their default output from **Get-ChildItem**. Most of the time **Get-ChildItem** returns a list of objects with the properties shown (in the two highlighted cases, however, it is not even returning objects, just a list of strings). In the case of file system space, every node in the tree is essentially equivalent to every other node with respect to its objects' types, either files or directories. In SQL Server space, however, almost every node is dealing with different objects; hence you should expect a description of those objects to vary.

Node	Default Properties
\	Name, Root, Description
\SQL	MachineName
\SQL*machine*	InstanceName
\SQL*machine**instance*	***list of object names***
\SQL*machine**instance*\Databases	Name, Status, RecoveryModel, CompatLvl, Collation, Owner
\SQL*machine**instance*\Databases*database*	***list of object names***
\SQL*machine**instance*\Databases*database*\Tables	Schema, Name, Created
\SQL*machine**instance*\Databases*database*\Views	Schema, Name, Created
\SQL*machine**instance*\Databases*database*\Roles	Name
\SQL*machine**instance*\Databases*database*\Triggers	Name, Created

The table lists the *default* properties displayed for each type of object when you use **Get-ChildItem** but the complete set is often much more vast. After completing discussion of navigation in SQL Server space with mapped drives, the next major section illustrates what additional information you can glean from **Get-ChildItem**.

Cmdlets Implemented by the SQL Server Provider

You have already seen examples of PowerShell cmdlets that are implicitly supported by the SQL Server provider: **Get-ChildItem** and **Set-Location**. There are several other important cmdlets supported, providing important functionality:

Cmdlet	Canonical alias	Other aliases	Description
Get-Location	gl	pwd	Gets current node
Set-Location	sl	cd, chdir	Changes current node
Get-ChildItem	gci	dir, ls	Lists the objects at current node
Get-Item	gi		Properties of current node
Rename-Item	rni	ren	Renames an object
Remove-Item	ri	del, erase, rd, rm, rmdir	Removes an object

Mapped Drives

Windows has had the concept of mapped drives for many generations, where you could provide a drive-letter alias to an arbitrary path, referencing your local file system or even a remote file system. This could be done from the Windows GUI, of course, but from the command-line you would use this command:

```
net use DriveLetter: \\ComputerName\Path
```

PowerShell provides an analogous command, but it allows you to create a mapped drive (or drive-letter alias) with any provider, not just the file system provider. The syntax for the SQL Server provider is:

```
New-PSDrive -Name name -PSProvider SQLSERVER -Root root
```

Unlike DOS drives, the drive name may be multiple characters (like the SQLSERVER: drive, for instance). And like many PowerShell commands, there are built-in aliases for the **New-PSDrive** cmdlet (`ndr` or `mount`). You have seen that the path to any objects within a database is rather lengthy. So use a command like this to make a shortcut:

```
mount sandboxDB SQLSERVER
SQLSERVER:\SQL\localhost\DEFAULT\Databases\sandbox
```

That command allows you to reference…

```
Get-ChildItem sandboxDB:\Tables
```

instead of…

```
Get-ChildItem
SQLSERVER:\SQL\localhost\DEFAULT\Databases\sandbox\Tables
```

Earlier you saw the output of the **Get-PSProvider** cmdlet listing current drives by provider. You can get a more detailed view of the drives themselves with the **Get-PSDrives** cmdlet:

Name	Provider	Root	Current Location
Alias	Alias		
C	FileSystem	C:\	usr\tmp
cert	Certificate	\	
D	FileSystem	D:\	
DB	SqlServer	SQLSERVER:\sql\localhost\SQLEXPRESS\Databases	sandbox
E	FileSystem	E:\	
Env	Environment		
F	FileSystem	F:\	
Function	Function		
HKCU	Registry	HKEY_CURRENT_USER	
HKLM	Registry	HKEY_LOCAL_MACHINE	
SQLSERVER	SqlServer	SQLSERVER:\	
Variable	Variable		
WSMan	WSMan		

I have highlighted the two drives on my system supported by the SQL Server provider, the default SQLSERVER: drive and my custom DB: drive. This cmdlet shows you what the drive is aliased to along with your current node location on that drive (i.e. the last location you navigated to with **Set-Location**).

Working in SQL Server Space

Understanding and Displaying Nodes

In the previous section you learned that in SQL Server space, unlike file system space, virtually every level of the hierarchy returns a different type of object. One table showed you the default properties of each object. But what object are we talking about? The table below enumerates the object type for each level. With the object type in hand, you can find its MSDN reference page to see documentation for all its properties.

Node	Object Type
\	Microsoft.SqlServer.Management.PowerShell.Extensions. SqlServerProviderExtension
\SQL	Microsoft.SqlServer.Management.PowerShell.Extensions.Machine
\SQL*machine*	Microsoft.SqlServer.Management.Smo.Server
\SQL*machine**instance*	*System.String*
\SQL*machine**instance*\Databases	Microsoft.SqlServer.Management.Smo.Database
\SQL*machine**instance*\Databases*database*	*System.String*
\SQL*machine**instance*\Databases*database*\Tables	Microsoft.SqlServer.Management.Smo.Table
\SQL*machine**instance*\Databases*database*\Views	Microsoft.SqlServer.Management.Smo.View
\SQL*machine**instance*\Databases*database*\Roles	Microsoft.SqlServer.Management.Smo.DatabaseRole
\SQL*machine**instance*\Databases*database*\Triggers	Microsoft.SqlServer.Management.Smo.Trigger

The object types were determined from this simple PowerShell sequence:

```
@(Get-ChildItem node)[0].GetType().FullName
```

For example:

```
@(Get-ChildItem SQLSERVER:\sql\localhost\sqlexpress\databases)[0].GetType().FullName
Microsoft.SqlServer.Management.Smo.Database
```

Each object may specify a *default* set of properties but this is often not the *complete* set of properties. The object type is again key to determining this default set of properties—and the default format (list or table)—for an object. First, PowerShell looks for a *formatting file*, which specifies both default properties and default formatting on individual .NET types. The main PowerShell system directory lists standard types but some modules, including **sqlps**, supplement this with additional formatting files. $HOME\Documents\WindowsPowerShell\Modules\sqlps\SQLProvider.Format.ps1xml defines the various database objects discussed here. If not specified in a formatting file, the default properties of an object will be gleaned from the **DefaultDisplayPropertySet** property if present. If that property is not defined, then there is no default set so all properties are displayed when an object is output. This selection process is summarized in the table.

Type specified in **module-specific** formatting file?	$env:windir\system32\ WindowsPowerShell \v1.0\Modules*. format.ps1xml $HOME\Documents\WindowsPowerShell\Modules*.format.ps1xml
Type specified in **system** formatting file?	$env:windir\system32\ WindowsPowerShell \v1.0*.format.ps1xml
DefaultDisplayPropertySet defined?	Use $*object*.PSStandardMembers.DefaultDisplayPropertySet; otherwise, display all properties

After the default property set is determined, the final determination of how to display objects is a choice between **Format-List** and **Format-Table**. You can always direct output to one of these specifically as the tail of your pipeline to mandate that format. Here's an example with Get-ChildItem to contrast these choices. This example uses the –Force parameter with a single wildcard argument (*), which forces display of all properties rather than just the default display properties. There are actually more columns in the output below; I have truncated the list to fit the page.

```
Get-ChildItem sqlserver:\sql | Format-Table -Force *
PSPath        PSParentPath  PSChildName  PSDrive     PSProvider  PSIsContainer
MachineName
------        ------------  -----------  -------     ----------  -------------
-----------
SqlServer:... SqlServer:... gandalf      SQLSERVER   SqlServer            True
gandalf
SqlServer:... SqlServer:... localhost    SQLSERVER   SqlServer            True
localhost
```

```
SqlServer:... SqlServer:... server509    SQLSERVER   SqlServer            True
server509
```

The truncation *within* each column, however, is not my doing! PowerShell does that in its attempt to squeeze as much into your current window width as possible because **Format-Table** puts an entire *record* on one line. By doing so, sometimes the data in a column is not terribly useful (the first and second columns, for example). In such cases where squeezing onto a line turns out to be impractical use Format-**List**, which displays one *property* per line, giving plenty of room to see the property's value. Here are the first couple objects above now given ample room:

```
Get-ChildItem sqlserver:\sql | Format-List -Force *
PSPath          : SqlServer::SQLSERVER:\sql\gandalf
PSParentPath    : SqlServer::SQLSERVER:\sql
PSChildName     : gandalf
PSDrive         : SQLSERVER
PSProvider      : SqlServer
PSIsContainer   : True
MachineName     : gandalf
ManagedComputer : Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer
Servers         : {[DEFAULT,
Microsoft.SqlServer.Management.PowerShell.Extensions.ServerInformation]}

PSPath          : SqlServer::SQLSERVER:\sql\localhost
PSParentPath    : SqlServer::SQLSERVER:\sql
PSChildName     : localhost
PSDrive         : SQLSERVER
PSProvider      : SqlServer
PSIsContainer   : True
MachineName     : localhost
ManagedComputer : Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer
Servers         : {[SQLEXPRESS,
Microsoft.SqlServer.Management.PowerShell.Extensions.ServerInformation]}

. . . (truncated)
```

If you do not direct your pipeline to one of the Format-verb cmdlets, PowerShell again looks to the same formatting files for direction. If the type is specified, it uses the specified format. If not found, then the final determination to use **Format-List** or **Format-Table** boils down to the number of properties to display: if more than four it uses **Format-List**, with four or less it uses **Format-Table**. Here's a summary of the steps:

Format-verb cmdlet specified in pipeline?	… | Format-List … … | Format-Table …
Type specified in **module-specific** formatting file?	$env:windir\system32\ WindowsPowerShell \v1.0\Modules*.format.ps1xml $HOME\Documents\WindowsPowerShell\Modules*.format.ps1xml
Type specified in **system** formatting file?	$env:windir\system32\ WindowsPowerShell \v1.0*.format.ps1xml
More than four properties?	Use **Format-List**; otherwise, use **Format-Table**

Finding Meta-Information: Row Counts

The **Invoke-SqlCmd** cmdlet described earlier is the workhorse to examine data in your database. But as either a developer or as a DBA, you often need to examine information

database. But as either a developer or as a DBA, you often need to examine information *about* your data, rather than the data itself. One common metric is how many records are in each table. There are many ways to determine this, as a quick web search will reveal. StackOverflow provides quite a few of them in answer to this one question: how to fetch the row count for all the tables in a SQL Server database. One of the most straightforward and concise, posted by Adrian Banks, is this:

```
CREATE TABLE #counts
(
    table_name VARCHAR(255),
    row_count INT
)

EXEC sp_MSForEachTable @command1='INSERT #counts (table_name,
row_count) SELECT ''?'', COUNT(*) FROM ?'
SELECT table_name, row_count FROM #counts ORDER BY table_name,
row_count DESC
```

This answer actually had the most votes at the time I wrote this but there are a number of reasons that it is less than optimal:

1. *Performance*: doing a SELECT COUNT(*) on huge tables is a tremendous performance hit; not only could it mean you have a long time to wait for an answer, you could potentially affect many other users by locking up tables for significant periods.
2. *Stability*: sp_MSForEachTable is an undocumented stored procedure, so you should not rely on it.
3. *Resource use*: it uses a temporary table; not a big thing, but a solution that does not use one would be better.

In contrast, one of the best in terms of performance and resources, posted by Keng, is this:

```
SELECT o.name,
  ddps.row_count
FROM sys.indexes AS i
  INNER JOIN sys.objects AS o ON i.OBJECT_ID = o.OBJECT_ID
  INNER JOIN sys.dm_db_partition_stats AS ddps ON i.OBJECT_ID =
ddps.OBJECT_ID
  AND i.index_id = ddps.index_id
WHERE i.index_id < 2  AND o.is_ms_shipped = 0 ORDER BY o.NAME
```

I submit, though, that this variation requires quite a bit of domain knowledge to craft from scratch and deriving it is far from obvious for many of us.

Now consider the same question in the context of PowerShell. From the above discussion you have seen that PowerShell makes a wealth of meta-information available on database objects with properties. In the current scenario, you know what you want (record counts) but do not know where to find it. It is not a huge leap to infer that a count of records for a table might be a property of tables. But what property? Use the **Get-Member** cmdlet to reveal all the properties for tables:

```
Get-ChildItem sandboxDB:\Tables | Get-Member -type properties
Name                      MemberType   Definition
----                      ----------   ----------
DisplayName               NoteProperty System.String DisplayName=dbo.BinaryDataTest
PSChildName               NoteProperty System.String PSChildName=dbo.BinaryDataTest
PSDrive                   NoteProperty System.Management.Automation.PSDriveInfo P...
PSIsContainer             NoteProperty System.Boolean PSIsContainer=True
```

```
PSParentPath              NoteProperty System.String PSParentPath=SqlServer::SQLS...
PSPath                    NoteProperty System.String PSPath=SqlServer::SQLSERVER:...
PSProvider                NoteProperty System.Management.Automation.ProviderInfo ...
AnsiNullsStatus           Property     System.Boolean AnsiNullsStatus {get;set;}
ChangeTrackingEnabled     Property     System.Boolean ChangeTrackingEnabled {get;...
Checks                    Property     Microsoft.SqlServer.Management.Smo.CheckCo...
. . . (truncated)
```

That yields a lengthy list of table properties. You could scan down the list and would likely notice one called `RowCount`. Or you could work smarter and ask for just the properties including the word **count**. The command sequence below gets a table object from Get-ChildItem, lists its members with Get-Member, then filters that list with Where-Object. For the purpose of formatting for this article I have used *aliases* for each cmdlet in the above sequence: **gci** for **Get-ChildItem**, **gm** for **Get-Member**, and **?** for **Where-Object**. (Aliases are quite useful when you are typing commands interactively as well!)

```
gci sandboxDB:\Tables | gm -type properties | ? { $_.name -match
'count' }
Name              MemberType Definition
----              ---------- ----------
RowCount          Property   System.Int64 RowCount {get;}
RowCountAsDouble  Property   System.Double RowCountAsDouble {get;}
```

This command reveals just what you need: a `RowCount` property for a table. With this in hand, it is trivial to list the number of records for each table. Get the table objects with **Get-ChildItem**, select the table name and the table row count with **Select-Object**, and pipe the results through **Format-Table** to provide a cleaner output:

```
Get-ChildItem sandboxDB:\Tables | Select-Object Name,RowCount |
Format-Table -AutoSize
```

That lists the row counts for every table; it is almost as simple to select just a few tables. In T-SQL you would just add a `WHERE` clause at a strategic point in the query. In PowerShell, you add another element to the pipeline, the same **Where-Object** used just above to filter the property list. This command assumes there are a number of tables containing the word `Big` in the name:

```
gci sandboxDB:\Tables | ?{$_.name -match 'Big'}| select Name,RowCount
| ft- AutoSize
```

Similarly, you could select based on schema with something like this:

```
gci sandboxDB:\Tables | where {$_.Schema -eq "dbo"} ...
```

Finding Meta-Information: Scripting DB Objects

Another task that you might want to tackle is to programmatically generate scripts to create your database. SQL Server Management Studio allows you to do this for any single database object with a context menu action, of course, whether it is an individual table or the database container itself. But doing this comprehensively for all your tables or, better still, for everything in your database, is non-trivial.

First consider the question of generating a script for one or more tables in your database. In fact, Stack Overflow presents this very question: How do I generate a CREATE TABLE statement for a given table? The answers posted there suggest a number of variations, all of

which are fairly complex. In contrast, here is the *complete* code to generate table creation scripts for *all* tables in your database (of course, substitute the appropriate path in SQL Server space to *your* database):

```
Get-ChildItem sandboxDB:\Tables | ForEach-Object { $_.Script() }
```

That is, you need merely invoke the `Script()` method on each Table object successively. That sequence displays output on the console. To send it to a file, add on one of several file output commands, as in:

```
gci sandboxDB:\Tables | % { $_.Script() } | Set-Content C:\create.sql
```

If you are a traditionalist who prefers your DDL in separate batches, use this to add the GO statement after each CREATE TABLE statement (you will find the line breaks are in all the right places):

```
gci sandboxDB:\Tables | % { $_.Script() + "GO" }
```

Here is a summary of these and other variations. I again use aliases here for brevity (**gci** for **Get-ChildItem**,**?** for **Where-Object**, and **%** for **ForEach-Object**):

Description	Command sequence
All tables (output to console)	**gci** sandboxDB:\Tables | **%** { $_.Script() }
All tables (output to file)	**gci** sandboxDB:\Tables | **%** { $_.Script() } | **Set-Content** C:\create.sql
All tables (separate batches)	**gci** sandboxDB:\Tables | **%** { $_.Script() + "GO" }
Selected tables	**gci** sandboxDB:\Tables | ? { $_.name -match "big.*" } | **%** { $_.Script() }
Single table	(**gci** sandboxDB:\Tables | ? { $_.name -eq "xyz_table" }).Script()

All tables with their indexes	**gci** sandboxDB:\Tables | **%** {
	$_.Script() + "GO"
	$_.Indexes | % { $_.Script() + "GO" }
	}

As you can appreciate, a very little PowerShell can accomplish quite a lot. But wait! Incredible as it may seem, this is actually the “low-level” way to script your database objects. At a higher level, you do not even need to worry about the “GO” statements, the associated objects, or even the looping. The brief script below, as you will observe, is mostly setting the various options on the **Scripter** object, plus a few introductory lines defining a couple variables. Beyond the bookkeeping all that remains is two lines of code, highlighted below:

```
$myScriptFile = "C:\usr\tmp\scripts.sql"                    # Specify
your output file
$myDbInstance = gi sqlserver:\sql\localhost\sqlexpress      # Specify
your server
$mydb = $myDbInstance.Databases["sandbox"]                  # Specify
your DB

$mydb.Script() | Out-File $myScriptFile     # <<<<< 1

$scrp = new-object ('Microsoft.SqlServer.Management.Smo.Scripter')
($myDbInstance)
$scrp.Options.AppendToFile = $True
```

```
$scrp.Options.ClusteredIndexes = $True
$scrp.Options.DriAll = $True
$scrp.Options.ScriptDrops = $False
$scrp.Options.IncludeIfNotExists = $True
$scrp.Options.IncludeHeaders = $True
$scrp.Options.ToFileOnly = $True
$scrp.Options.Indexes = $True
$scrp.Options.WithDependencies = $True
$scrp.Options.FileName = $myScriptFile

$scrp.Script([Microsoft.SqlServer.Management.Smo.SqlSmoObject[]]$mydb.Tables)    #
<<<<< 2
```

The first line calls the `Script()` method on the database itself to generate the CREATE DATABASE code. The second line calls the `Script()` method on the **Scripter** object, passing in the set of tables in the database. The **Scripter** object is Microsoft's top level object for managing scripting operations—see the MSDN reference page. The power of this class comes from its **Options** property, where you configure what it will do when you execute. The most important option is **WithDependencies**, which lets you automatically include all dependent objects in the generated script. Note that the first line creates or overwrites the specified file, giving you a clean slate, so to speak. The second line adds to that because the **AppendToFile** option is enabled. See all the options on the MSDN ScriptingOptions page.

This script is an adaptation of Edwin Sarmiento's code in his blog post Generating SQL Scripts using Windows PowerShell, with two minor bug fixes. His article provides an excellent discussion of scripting DB objects with PowerShell for supplemental reading.

Conclusion: The Spectrum of PowerShell / SQL Server Entities

This article covered the full spectrum of SQL Server capabilities in PowerShell with broad strokes, providing a plethora of practical tips and techniques for making you productive quickly. The table below summarizes the concepts you have seen thus far and adds one more important one, SQLPSX, for exploration on your own.

SMO	Short for **SQL Server Management Objects**, SMO is a set of .NET classes to create applications that manage SQL Server. All of the nodes in SQL Server space discussed herein resolve to SMO objects (e.g. database tables are Microsoft.SqlServer.Management.Smo.Table objects). SMO was introduced in SQL Server 2005.
SQLPS	This PowerShell module provides (1) several new cmdlets—notably Invoke-SqlCmd—that allow you to interact with a database, and (2) the SQL Server provider with which you can navigate SQL Server space. This module was included with SQL Server 2012 but, thanks to an eponymous package wrapped up by Chad Miller, may just as easily be used with earlier SQL Server editions. (Not to complicate the issue, there is also a **sqlps** utility provided in SQL Server 2008 that runs a "mini" PowerShell with SQL support, but this utility is deprecated in favor of the **sqlps** module.)
SQL Server Provider	This provider allows you to interact with the hierarchy of SQL Server objects just as a native PowerShell file system provider allows you to interact with files. You can navigate through SQL Server space using paths analogously to file paths. As MSDN explains, "You can use the paths to locate an object, and then use methods from the SQL Server Management Object (SMO) models to perform actions on the objects."
SQLPSX	This is Chad Miller's CodePlex project first published in 2008, prior to SQL Server having any PowerShell support. Despite the later support added by Microsoft via the **sqlps** module, SQLPSX's wide range of SQL-related functions still provide functionality not covered by **sqlps**. Quoting the home page of the project, "SQLPSX consists of 13 modules with 163 advanced

> functions, 2 cmdlets and 7 scripts for working with ADO.NET, SMO, Agent, RMO, SSIS, SQL script files, PBM, Oracle and MySQL and using PowerShell ISE as a SQL and Oracle query tool."

(Both the StackOverflow question What's the difference between PowerShell / SQL Server snap in's / tools? and Chad Miller's answer to it provided my inspiration for organizing this conclusion.)

Now that you've got a good foundation in PowerShell for SQL Server from this article, it is time to go out into the real world. A great place to start is PowerShell SMO: Just Writing Things Once, penned by the prolific Phil Factor, which walks you through real scenarios (with code) for managing multiple databases.

Author profile: Michael Sorens

Michael Sorens is passionate about software to be more productive, evidenced by his open source libraries in several languages (see his API bookshelf) as well as SqlDiffFramework (a DB comparison tool for heterogeneous systems including SQL Server, Oracle, and MySql). With degrees in computer science and engineering he has worked the gamut of companies from Fortune 500 firms to Silicon Valley startups over the last 25 years or so. Current passions include PowerShell, .NET, SQL, and XML technologies (see his full brand page). Spreading the seeds of good design wherever possible, he enjoys sharing knowledge via writing (see his full list of articles), teaching, and StackOverflow. Like what you have read? Connect with Michael on LinkedIn and Google +

Search for other articles by Michael Sorens