http://www.sqlservercentral.com/articles/Stairway+Series/109321/
Printed 2014/05/06 09:54PM

Stairway to SQL PowerShell Level 6: PowerShell Modules

By Ben Miller, 2014/04/16

In this level, we will walk through creating and using modules for PowerShell. Modules are simply a collection of functions that are most likely related to each other and packaged in a way that allow you to import them as a group. After they are imported, you can use them like cmdlets or commands in scripts or interactively. Modules that you create, as well as others that you might find out on the web or shared among friends, can be loaded manually or through your profile to make them available to any of your PowerShell sessions.

Basics of Modules

We'll begin with the components of a module. Modules are collections of functions put together in a file or files to be imported as a group for later use. The basic components are listed in Table 6.1 below. The only one that is required is the .psm1 file the others are advanced items and will not be discussed in this level.

Component	Description
.psd1 file	The psd1 file is called the "Manifest".
.psm1 file	The psm1 file is referred to as the script module file. This file will contain the functions that the module will export for use in scripts or interactively.
.ps1xml file	This type of file is the formatting and type file. The use of this file is to control the way the objects are formatted when output in the PowerShell environment. See http://technet.microsoft.com/en-us/library/hh847831.aspx.
.dll file	This is an assembly file that may contain functions or cmdlets that can be loaded into the environment. These files are compiled .NET code and not merely script.

Table 6.1 Basic components that make a module

We will focus this level on the basics of writing modules. What you will see in the example modules in this level are PowerShell functions that are exported for use in that PowerShell session. The command to interact with modules is `Import-Module`. This command takes the module name as a parameter, and the functions that make up the specified module will be imported into the current session to be used. There are other parameters to the Import-Module command such as *DisableNameChecking* for those modules that contain function names that are not conformant to the approved PowerShell verbs. There is a list of approved verbs to use in the verb-noun naming convention in PowerShell, and you can find a comprehensive list here: http://msdn.microsoft.com/en-us/library/windows/desktop/ms714428(v=vs.85).aspx. In Listing 6.1 you will see a module called *TestSqlModule*. Do not run this script interactively; it must be loaded and run from a script file. That process will be described shortly.

```
<#
.SYNOPSIS
Gets a ServerConnection.
.DESCRIPTION
The Get-SqlConnection function  gets a ServerConnection to the specified SQL Server.
.INPUTS
None
    You cannot pipe objects to Get-SqlConnection
.OUTPUTS
Microsoft.SqlServer.Management.Common.ServerConnection
    Get-SqlConnection returns a Microsoft.SqlServer.Management.Common.ServerConnection object.
.EXAMPLE
Get-SqlConnection "localhost\instancename"
This command gets a ServerConnection to SQL Server localhost\instancename using Windows Authentication.
.EXAMPLE
Get-SqlConnection " localhost\instancename " "sa" "Passw0rd"
This command gets a ServerConnection to SQL Server localhost\instancename using SQL authentication.
.LINK
Get-SqlConnection
#>
function Get-SqlConnection
{
    param(
    [Parameter(Position=0, Mandatory=$true)] [string]$sqlserver,
    [Parameter(Position=1, Mandatory=$false)] [string]$username,
    [Parameter(Position=2, Mandatory=$false)] [string]$password
    )

    if($Username -and $Password)
    { $con = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver,$username,$password }
```

```
    else
    { $connection = new-object ("Microsoft.SqlServer.Management.Common.ServerConnection") $sqlserver }

    $connection.Connect()

    Write-Output $connection

} #Get-SqlConnection


Export-ModuleMember -function Get-SqlConnection
```

Listing 6.1 Example TestSqlModule.psm1 code

Listing 6.1 contains a complete function including comments. The reason is can't be run interactively is because of the *Export-ModuleMember* cmdlet which you can see at the end of the listing. This cmdlet is only allowed inside a .psm1 module file module file. This cmdlet is the one that allows the definitions in the module to be later imported into your PowerShell session. To be able to import this module via I*mport-Module* you need to save the complete Listing 6.1 in a file with a *.psm1* extension. The file should be located in a directory under the *modules* directory. In other words, to create a *TestSqlModule* module, I would need to create a folder called *TestSqlModule* under the Modules directory, either in my private documents *WindowsPowerShell\Modules* directory or in the Windows or global location referenced in Listing 6.2. Once the *TestSqlModule.psm1* file (containing the code in Listing 6.1) is saved inside that new directory, I could import it into my PowerShell session using the code in Listing 6.2. The last line of Listing 6.2 lists all the available modules, and you should see your new *TestSqlModule* in the output list.

```
# Command to import a module named TestSqlModule
# PowerShell will look in My Documents\WindowsPowerShell\Modules\TestSqlModule for a file called TestSqlModule.psm1
# Or in the C:\Windows\System32\WindowsPowerShell\v1.0\Modules\TestSqlModule directory

Import-Module TestSqlModule

# To list modules available to load or import
Get-Module -ListAvailable
```

Listing 6.2 Loading TestSqlModule into your PowerShell session

Let's take a look at the pieces in Listing 6.1. First in the listing is the comment section. This is the syntax you use when documenting the function. The syntax includes sections beginning with a period (.) and a keyword. Listed in table 6.2 are the main sections for documenting purposes that are picked up by the Get-Help cmdlet. To get a full description of how the help documentation of a function works, type `help about_comment_based_help` in a PowerShell window.

Keyword	Explanation
SYNOPSIS	Contains a synopsis of the function or script to understand what this function or script actually does and what you want the user to know in a summary.
DESCRIPTION	A lengthier piece of documentation that typically goes a little more in depth than the SYNOPSIS section.
PARAMETER	Parameter syntax allows multiple parameters to be illustrated in the documentation and it is important to use this if your function or script has parameters and that they have an explanation connected to understand how this parameter is used, what order it is in.
EXAMPLE	Example syntax also allows multiple examples. This is the script developers chance to show examples of how to use this function or script. It is text and you can do a lot in the example syntax. Key part of the documentation.
NOTES	Contains other notes to the user to ensure enough information is given so allow success when using this function or script.
INPUTS	Contains a reference to the inputs that you can send to this function.
OUTPUTS	Documents the outputs that come from the function.

Table 6.2 Comment sections for documentation of PowerShell functions

Where do modules live?

There are two places PowerShell looks for Modules when using the ***Import-Module*** cmdlet. One is in the c:\windows\system32\WindowsPowerShell\v1.0\Modules directory and the other is in your documents directory in the WindowsPowerShell\Modules directory, which is the one shown in Listing 6.2. These two places are used because you may have modules that you want to live in a global place and not rely on your profile documents directory. If you want modules in a private place, then your profile documents directory works great. One thing to note, when using the cmdlet *Import-Module*, it will look for the module first from your profile documents directory and if not found there, it will look in

the global c:\windows\system32\WindowsPowerShell\v1.0\Modules directory. If it finds the module in your personal documents folder, it will load it from there and will not look in the global directory.

As discussed in a previous level, functions are reusable code that allow you to call them with parameters to perform operations in a consistent manner. To use these functions that are in a script, you would *dot-source* it. An example of a file that can be dot-sourced is in Listing 6.3

```
# dot sourcing a file with a function to reuse the function
# save the following code into a file called afunction.ps1
Function Get-CurrentDirListing {
   Dir
}
```

Listing 6.3 Creating a file that will be used with dot-sourcing

Dot-sourcing simply means to use a period and then a space followed by the name of a .ps1 file to load in.

If you've created the file suggested in Listing 6.3, you can now go to a command line in PowerShell and run the following starting with the .

```
C:\> . c:\powershell\afunction.ps1
```

If the *afunction.ps1* file exists and contains a function called *Get-CurrentDirListing*, and you dot-sourced it, then you could call *Get-CurrentDirListing* from a PowerShell prompt. Dot-sourcing is merely loading the script into the PowerShell environment without executing the script and the function and variables get added to the global scope. When using modules, you have more control as the variables are all automatically private and with the use of *Export-ModuleMember* you can decide which components you want to export, variables and/or functions, to the public space. Modules can also be removed by *using Remove-Module* where dot-sourcing a .ps1 file is global and can be replaced but not removed. The module is imported into your session via *Import-Module* and the functions that are exported via *Export-ModuleMember* inside the module are now available in your session for use. You can find out which functions were or will be loaded in an import operation by using `Get-Command -Module ModuleName`. Just remember that *Import-Module* is what you use to include them in your session and the module must exist, and the cmdlet *Export-ModuleMember* is used inside a .psm1 file to export the function and variables to the session. If you do not specify *Export-ModuleMember* for each function, you will not be able to execute the cmdlet or function in your session.

Introducing SQLPSX, a Module for SQL Server

As mentioned, there are PowerShell modules that can be found on the internet and downloaded for use. There is a module set called SQLPSX at http://sqlpsx.codeplex.com. It is deployed as an MSI file which is an installer that will put the modules in your personal or private modules folder. It was written as a project by Chad Miller and others that contains many modules and many functions in those modules that are used to access SQL Server objects with a robust framework. The contents of SQLPSX provide you with some great functions to get to SQL Server connections, databases, tables, etc. Listing 6.4 gives an example of loading SQLPSX and then using a function to get an SMO Database object from SQL Server by specifying the server name and the database name. Figure 6.1 shows the output when loading the module and the output of getting a database.

```
Import-Module SQLPSX

# localhost\I12 is my instance name, but you can substitute an instance that you have to test this.
$db = Get-SqlDatabase -sqlserver localhost -dbname master
$db.Name
```

Listing 6.4 Import Module SQLPSX and get a database

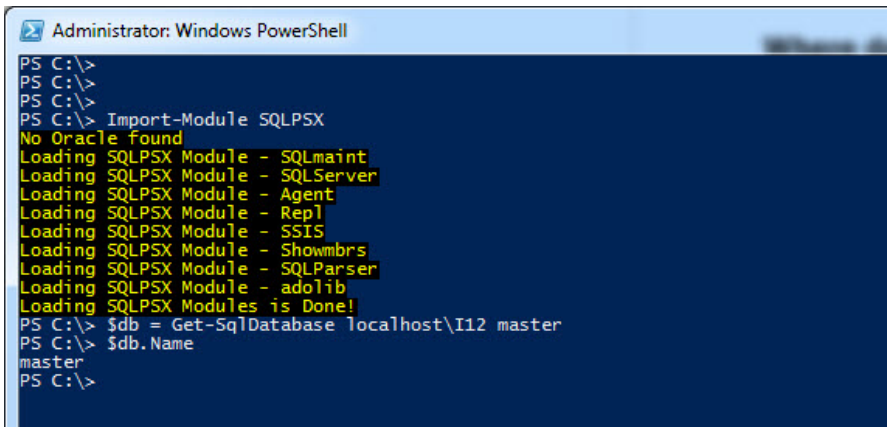

Figure 6.1 Output from loading SQLPSX and using Get-SqlDatabase

As you can see from the listing and the figure above, it became very simple to get a SQL Database object from SQL Server with just the name of the server and the database name. This is the power of reusable functions or in this case a function loaded by loading a module.

Summary

Modules are a powerful way to package functions that are related to each other and load them either manually or in your profile. We only covered

script based modules, but if you look here http://msdn.microsoft.com/en-us/library/windows/desktop/dd901839(v=vs.85).aspx you will find that there are more kinds of modules and more in depth information about creating modules. The important thing to remember is that when packaged as a module you can distribute or install these on servers and make use of them across the enterprise.
