1

Getting Started with SQL Server and PowerShell

In this chapter, we will cover:

- Working with the sample code
- Exploring the SQL Server PowerShell hierarchy
- Installing SMO
- Loading SMO assemblies
- Discovering SQL-related cmdlets and modules
- Creating a SQL Server instance object
- Exploring SMO Server objects

Introduction

PowerShell is an administrative tool that has both shell and scripting capabilities that can leverage **Windows Management Instrumentation** (**WMI**), COM components, and .NET libraries. PowerShell is becoming more prominent with each generation of Microsoft products. Support for it is being bundled, and improved, in a number of new and upcoming Microsoft product releases. Windows Server, Exchange, ActiveDirectory, SharePoint, and even SQL Server, have all shipped with added PowerShell support and cmdlets. Even vendors such as VMWare, Citrix, Cisco, and Quest, to name a few, have provided ways to allow their products to be accessible via PowerShell.

What makes PowerShell tick? Every systems administrator probably knows the pain of trying to integrate heterogeneous systems using some kind of scripting. Historically, the solution involved some kind of VBScript, some good old batch files, maybe some C# code, some Perl—you name it. Sysadmins either had to resort to duct taping different languages together to get things to work the way they intended, or just did not bother because of the complicated code.

This is where PowerShell comes in. One of the strongest points for PowerShell is that it simplifies automation and integration between different Microsoft *ecosystems*. As most products have support for PowerShell, getting one system to talk to another is just a matter of discovering what cmdlets, functions, or modules need to be pulled into the script. Even if the product does not have support yet for PowerShell, it most likely has .NET or COM support, which PowerShell can easily use.

Notable PowerShell V3 features

Some of the notable features in the latest PowerShell version are:

- **Workflows**: PowerShell V3 introduces **Windows PowerShell Workflow** (**PSWF**), which as stated in MSDN (`http://msdn.microsoft.com/en-us/library/jj134242.aspx`):

  > *helps automate the distribution, orchestration, and completion of multi-computer tasks, freeing users and administrators to focus on higher-level tasks.*

  PSWF leverages Windows Workflow Foundation 4.0 for the declarative framework, but using familiar PowerShell syntax and constructs.

- **Robust sessions**: PowerShell V3 supports more robust sessions. Sessions can now be retained amid network interruptions. These sessions will remain open until they time out.
- **Scheduled jobs**: There is an improved support for scheduled tasks. There are new cmdlets in the `PSScheduledJob` module that allow you to create, enable, and manage scheduled tasks.
- **Module AutoLoading**: If you use a cmdlet that belongs to a module that hasn't been loaded yet, this will trigger PowerShell to search `PSModulePath` and load the first module that contains that cmdlet. This is something we can easily test:

```
#check current modules in session
Get-Module

#use cmdlet from CimCmdlets module, which
#is not loaded yet
Get-CimInstance win32_bios

#note new module loaded CimCmdlets
Get-Module

#use cmdlet from SQLPS module, which
#is not loaded yet
Invoke-Sqlcmd -Query "SELECT GETDATE()" -ServerInstance "KERRIGAN"

#note new modules loaded SQLPS and SQLASCmdlets
Get-Module
```

- **Web service support**: PowerShell V3 introduces the `Invoke-WebRequest` cmdlet, which sends HTTP or HTTPS requests to a web service and returns the object-based content that can easily be manipulated in PowerShell. You can think about downloading entire websites using PowerShell (and check out Lee Holmes' article on it: `http://www.leeholmes.com/blog/2012/03/31/how-to-download-an-entire-wordpress-blog/`).
- **Simplified language syntax**: Writing your `Where-Object` and `Foreach-Object` has just become cleaner. Improvements in the language include supporting default parameter values, and simplified syntax.

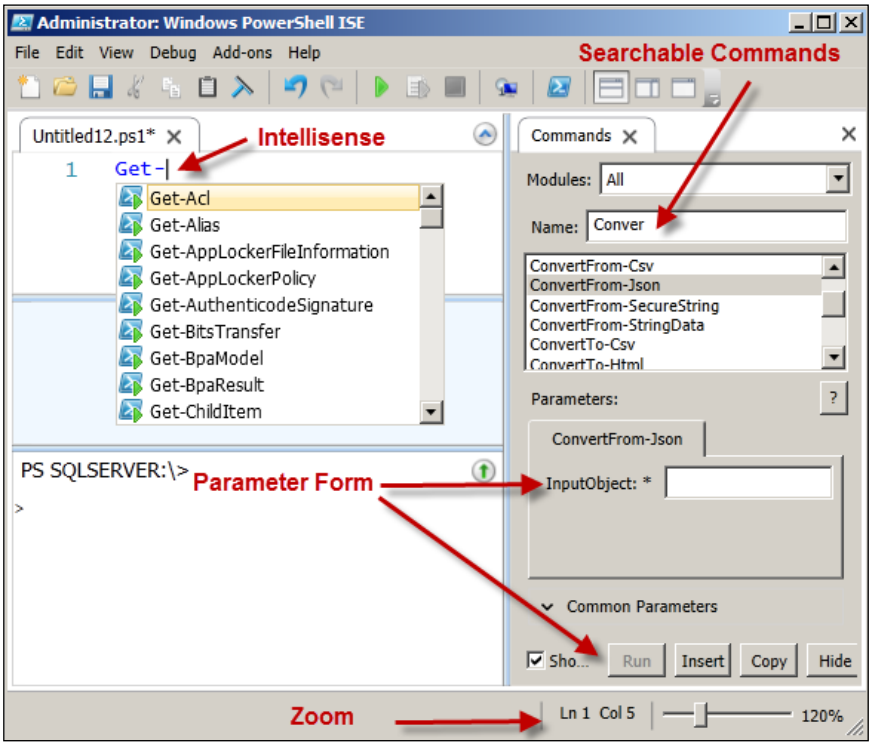
  What you used to write in V1 and V2 with curly braces and `$_` as follows:

  ```
  Get-Service | Where-Object { $_.Status -eq 'Running' }
  ```

  can now be rewritten in V3 as:

  ```
  Get-Service | Where-Object Status -eq 'Running'
  ```

- **Improved Integrated Scripting Environment** (**ISE**): The new ISE comes with Intellisense, searchable commands in the sidebar, parameter forms, and live syntax checking.



Before you start: Working with SQL Server and PowerShell

Before we dive into the recipes, let's go over a few important concepts and terminologies that will help you understand how SQL Server and PowerShell can work together:

- **PSProvider and PSDrive**: PowerShell allows different data stores to be accessed as if they are regular files and folders. `PSProvider` is similar to an adapter, which allows these data stores to be seen as drives.

  To get a list of the supported `PSProvider` objects, type:

  ```
  Get-PSProvider
  ```

You should see something similar to the following screenshot:

```
Name                Capabilities                    Drives
----                ------------                    ------
Alias               ShouldProcess                   {Alias}
Environment         ShouldProcess                   {Env}
FileSystem          Filter, ShouldProcess           {C, A, D}
Function            ShouldProcess                   {Function}
Registry            ShouldProcess, Transactions     {HKLM, HKCU}
Variable            ShouldProcess                   {Variable}
Certificate         ShouldProcess                   {Cert}
WSMan               Credentials                     {WSMan}
SqlServer           Credentials                     {SQLSERVER}
```

Depending on which instance of `PSProvider` is already available in your system, yours may be slightly different:

- **PSDrive**: Think of your `C:\`, but for data stores other than the file system. To get a list of `PSDrive` objects in your system, type:

  ```
  Get-PSDrive
  ```

  You should see something similar to the following screenshot:

  ```
  Name           Used (GB)     Free (GB) Provider      Root
  ----           ---------     --------- --------      ----
  A                                      FileSystem    A:\
  Alias                                  Alias
  C                  46.18         33.72 FileSystem    C:\
  Cert                                   Certificate   \
  D                                      FileSystem    D:\
  Env                                    Environment
  Function                               Function
  HKCU                                   Registry      HKEY_CURRENT_USER
  HKLM                                   Registry      HKEY_LOCAL_MACHINE
  SQLSERVER                              SqlServer     SQLSERVER:\
  Variable                               Variable
  WSMan                                  WSMan
  ```

  Note that there is a `PSDrive` for SQLServer, which can be used to navigate, access, and manipulate SQL Server objects.

- **Execution policy**: By default, PowerShell will abide by the current execution policy to determine what kind of scripts can be run. For our recipes, we are going to assume that you will run PowerShell as the administrator on your test environment. You will also need to set the execution policy to `RemoteSigned`:

  ```
  Set-ExecutionPolicy RemoteSigned
  ```

  This setting will allow PowerShell to run digitally-signed scripts, or local unsigned scripts.

- **Modules and snap-ins**: Modules and snap-ins are ways to extend PowerShell. Both modules and snap-ins can add cmdlets and providers to your current session. Modules can additionally load functions, variables, aliases, and other tools to your session.

Snap-ins are **Dynamically Linked Libraries** (**DLL**), and need to be registered before they can be used. Snap-ins are available in V1, V2, and V3. For example:

```
Add-PSSnapin SqlServerCmdletSnapin100
```

Modules, on the other hand, are more like your regular PowerShell `.ps1` script files. Modules are available in V2 and V3. You do not need to register a module to use it, you just need to import:

```
Import-Module SQLPS
```

For more information on PowerShell basics, check out *Appendix B*, *PowerShell Primer*.

Working with the sample code

Samples in this book have been created and tested against SQL Server 2012 on Windows Server 2008 R2.

To work with the sample code in this book using a similar VM setup that the book uses, see *Appendix D*, *Creating a SQL Server VM*.

How to do it...

If you want to use your current machine without creating a separate VM, as illustrated in *Appendix D*, *Creating a SQL Server VM*, follow these steps to prepare your machine:

1. Install SQL Server 2012 on your current operating system—either Windows 7 or Windows Server 2008 R2. See the list of supported operating systems for SQL Server 2012:

   ```
   http://msdn.microsoft.com/en-us/library/ms143506.aspx
   ```

2. Install PowerShell V3.

   Although PowerShell V3 comes installed with Windows 8 and Windows Server 2012, at the time of writing this book these two operating systems are not listed in the list of operating systems that SQL Server 2012 supports.

   To install PowerShell V3 on Windows 7 SP1, Windows Server 2008 SP2, or Windows Server 2008 R2 SP1:

   Install Microsoft .NET Framework 4.0, if it's not already there.

Download and install Windows Management Framework 3.0, which contains PowerShell V3. At the time of writing this book, the **Release Candidate** (**RC**) is available from:

`http://www.microsoft.com/en-us/download/details.aspx?id=29939`

3. Enable PowerShell V3 ISE. We will be using the improved Integrated Scripting Environment in many samples in this book:
   - Right-click on **Windows PowerShell** on your taskbar and choose **Run as Administrator**.
   - Execute the following:

     ```
     PS C:\Users\Administrator>Import-Module ServerManager PS C:\
     Users\Administrator>Add-WindowsFeature PowerShell-ISE
     ```

   - Test to ensure you can see and launch the ISE:

     ```
     PS C:\Users\Administrator> powershell_ise
     ```

     Alternatively you can go to **Start** | **All Programs** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.

   - Set execution policy to `RemoteSigned` by executing the following, on the code editor:

     ```
     Set-ExecutionPolicy RemoteSigned
     ```

If you want to run PowerShell V2 and V3 side by side, you can check out Jeffery Hicks' article, *PowerShell 2 and 3, Side by Side*:

`http://mcpmag.com/articles/2011/12/20/powershell-2-and-3-side-by-side.aspx`

See also

- Check out the PowerShell V3 Sneak Peek Screencast:

  `http://technet.microsoft.com/en-us/edge/Video/hh533298`

- See also the SQL Server PowerShell documentation on MSDN:

  `http://msdn.microsoft.com/en-us/library/hh245198(SQL.110).aspx`

Exploring the SQL Server PowerShell hierarchy

In SQL Server 2012, the original mini-shell has been deprecated, and SQLPS is now provided as a module. Launching PowerShell from SSMS now launches a Windows PowerShell session, imports the `SQLPS` module, and sets the current context to the item the PowerShell session was launched from. DBAs and developers can then start navigating the object hierarchy from here.

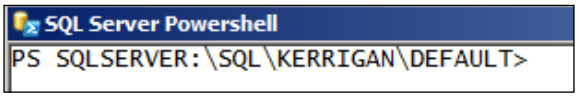
Getting ready

Log in to SQL Server 2012 Management Studio.

How to do it...

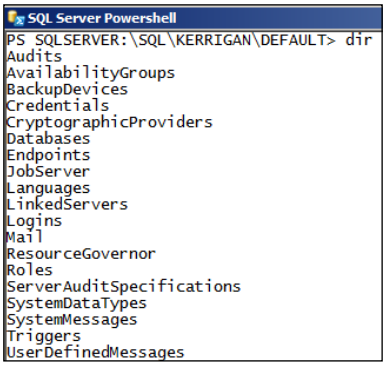
In this recipe, we will navigate the SQL Server PowerShell hierarchy by launching a PowerShell session from SQL Server Management Studio:

1. Right-click on your instance node.
2. Click on **Start PowerShell**. This will launch a PowerShell session and load the `SQLPS` module. This window looks similar to a command prompt, with a prompt set to the SQL Server object you launched this window from:

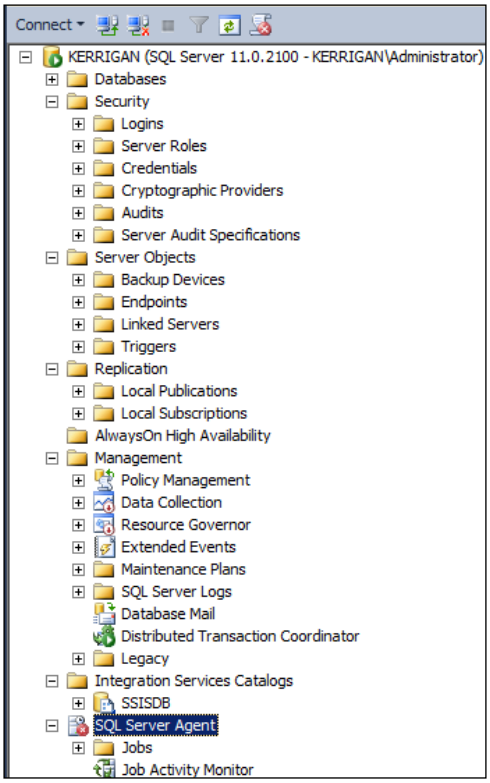


   Note the starting path in this window.

3. Type `dir`. This should give you a list of all objects directly accessible from the current server instance—in our case, from the default SQL Server instance **KERRIGAN**. Note that **dir** is an alias for the cmdlet `Get-ChildItem`.



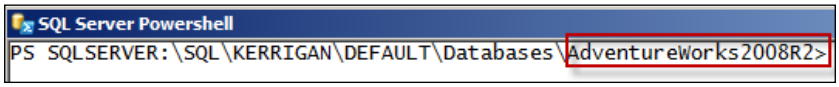
This is similar to the objects you can find under the instance node in **Object Explorer** in SQL Server Management Studio.

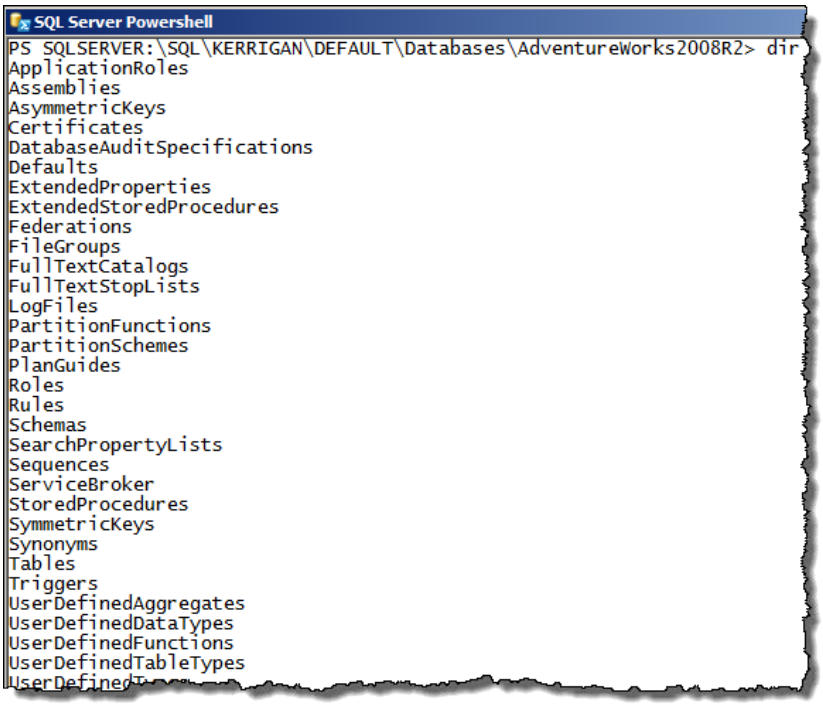


4. While our PowerShell window is open, let's explore the SQL Server `PSDrive`, or the SQL Server data store, which PowerShell treats as a series of items. Type `cd\`. This will change the path to the root of the current drive, which is our SQL Server `PSDrive`.
5. Type `dir`. This will list all Items accessible from the root SQL Server `PSDrive`. You should see something similar to the following screenshot:

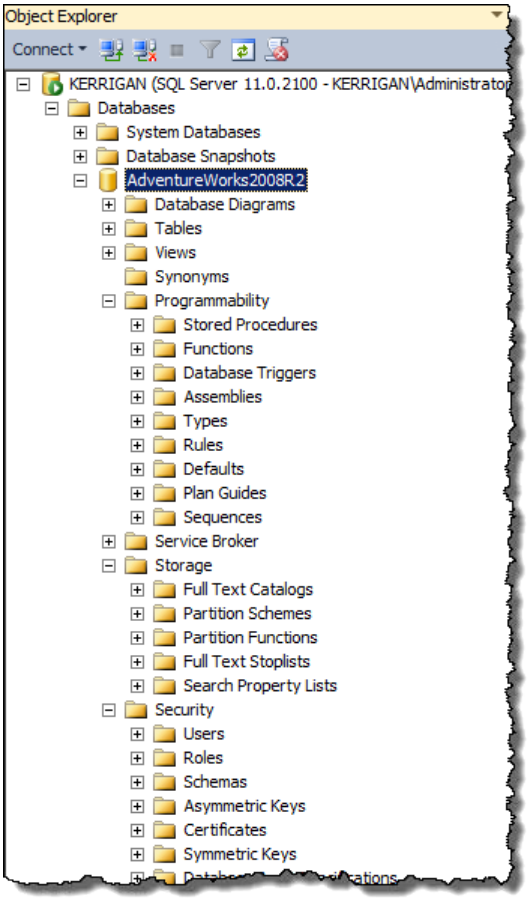
```
SQL Server Powershell
PS SQLSERVER:\SQL\KERRIGAN\DEFAULT> cd \
PS SQLSERVER:\> dir

Name            Root                          Description
----            ----                          -----------
SQL             SQLSERVER:\SQL                SQL Server Database Engine
SQLPolicy       SQLSERVER:\SQLPolicy          SQL Server Policy Management
SQLRegistration SQLSERVER:\SQLRegistration    SQL Server Registrations
DataCollection  SQLSERVER:\DataCollection     SQL Server Data Collection
XEvent          SQLSERVER:\XEvent             SQL Server Extended Events
Utility         SQLSERVER:\Utility            SQL Server Utility
DAC             SQLSERVER:\DAC                SQL Server Data-Tier Application Compone
                                              nt
IntegrationServ SQLSERVER:\IntegrationServices SQL Server Integration Services
ices
SQLAS           SQLSERVER:\SQLAS              SQL Server Analysis Services
```

6. Close this window.
7. Go back to **Management Studio**, and right-click on one of your user databases.
8. Click on **Start PowerShell**. Note that this will launch another PowerShell session, with a path that points to the database you right-clicked from:





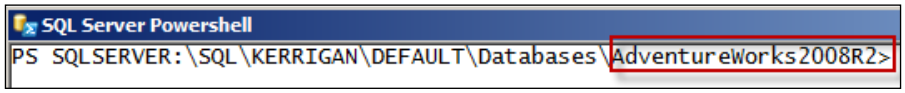
Note that the starting path of this window is different from the starting path when you first launched PowerShell in the second step. If you type `dir` from this location, you will see all items that are sitting underneath the **AdventureWorks2008R2** database.



You can see some of the items enumerated in this screen in SQL Server Management Studio's **Object Explorer**, if you expand the **AdventureWorks2008R2** database node.

How it works...

When PowerShell is launched through Management Studio, a *context-sensitive* PowerShell session is created, which automatically loads the `SQLPS` module. This will be evident in the prompt, which by default shows the current path of the object from which the **Start PowerShell** menu item was clicked.



SQL Server 2008/2008 R2 was shipped with a SQLPS mini-shell, also referred to as SQLPS utility. This can also be launched from SSMS by right-clicking on an object from **Object Explorer**, and clicking on **Start PowerShell**. This mini-shell was designed to be a closed shell preloaded with SQL Server extensions. This shell was meant to be used for SQL Server only, which proved to be quite limiting because DBAs and developers often need to load additional snap-ins and modules in order to integrate SQL Server with other systems through PowerShell. The alternative way is to launch a full-fledged PowerShell session, and depending on your PowerShell version either load snap-ins or load the `SQLPS` module.

In SQL Server 2012, the original mini-shell has been deprecated. When you launch a PowerShell session from SSMS in SQL Server 2012, it launches the full-fledged PowerShell session, with the updated `SQLPS` module loaded by default.

SQL Server is exposed as a PowerShell drive (`PSDrive`), which allows for traversing of objects as if they are folders and files. Thus, familiar commands for traversing directories are supported in this provider, such as `dir` or `ls`. Note that these familiar commands are often just aliases to the real cmdlet name, in this case, `Get-ChildItem`.

When you launch PowerShell from Management Studio, you can immediately start navigating the SQL Server PowerShell hierarchy.

Installing SMO

SQL Server Management Objects (**SMO**) was introduced with SQL Server 2005 to allow SQL Server to be accessed and managed programmatically. SMO can be used in any .NET language, including C#, VB.NET, and PowerShell. SMO is the key to automating most SQL Server tasks. SMO is also backward compatible to previous versions of SQL Server, extending support all the way to SQL Server 2000.

SMO is comprised of two distinct classes: instance classes and utility classes.

Instance classes are the SQL Server objects. Properties of objects such as the server, the databases, and tables can be accessed and set using the instance classes.

Utility classes are helper or utility classes that accomplish common SQL Server tasks. These classes belong to one of three groups: Transfer class, Backup and Restore classes, or Scripter class.

To gain access to the SMO libraries, SMO needs to be installed, and the SQL Server-related assemblies need to be loaded.

Getting ready

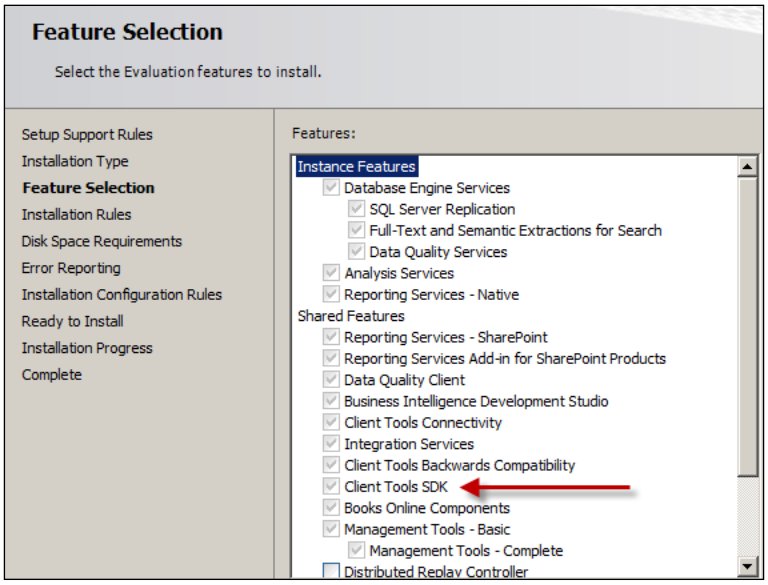
There are a few ways to get SMO installed:

- If you are installing SQL Server 2012, or already have SQL Server 2012, SMO can be installed by installing **Client Tools SDK**. Get your install disk or image ready.
- If you want just SMO installed without installing SQL Server, download the SQL Server Feature 2012 pack.

How to do it...

If you are installing SQL Server or already have SQL Server:

1. Load up your SQL Server install disk or image, and launch the `setup.exe` file.
2. Select **New SQL Server standalone installation or add features to an existing installation**.
3. Choose your installation type, and click on **Next**.
4. In the **Feature Selection** window, make sure you select **Client Tools SDK**.



5. Complete your installation.

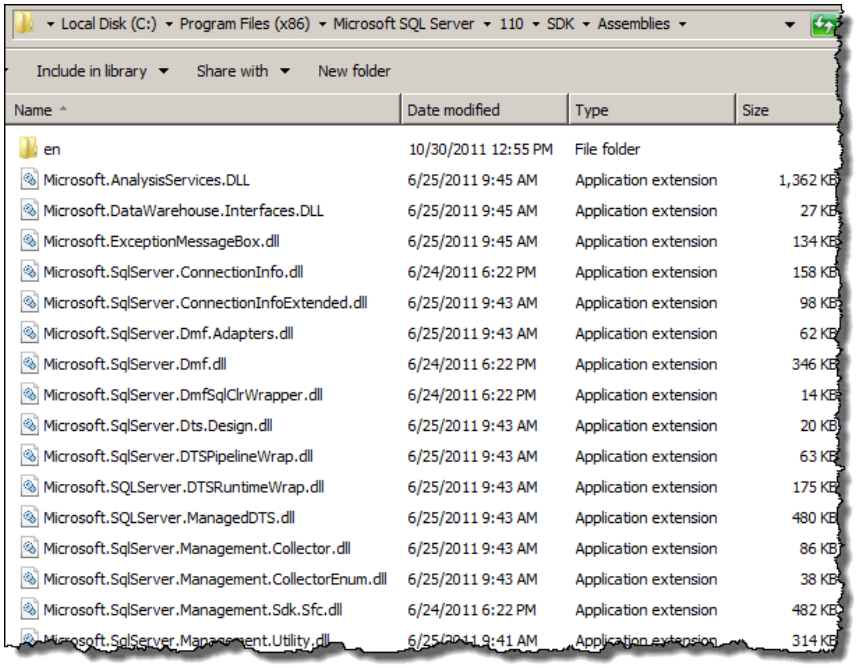
After this, you should already have all the binaries needed to use SMO.

If you are not installing SQL Server, you must install SMO using the SQL Server Feature Pack on the machine you are using SMO with:

1. Open your web browser, go to your favorite search engine, and search for SQL Server 2012 Feature Pack.
2. Download the package.
3. Double-click on **SharedManagementObjects.msi** to install.

There's more...

By default, the SMO assemblies will be installed in **<SQL Server Install Directory>\110\SDK\Assemblies**.



Loading SMO assemblies

Before you can use the SMO library, the assemblies need to be loaded. In SQL Server 2012, this step is easier than ever.

Getting ready

SQL Management Objects(**SMO**) must have already been installed on your machine.

How to do it...

In this recipe, we will load the `SQLPS` module.

1. Open up your PowerShell console, or PowerShell ISE, or your favorite PowerShell editor.
2. Type the import-module command as follows:

   ```
   Import-Module SQLPS
   ```

3. Confirm that the module is loaded:

   ```
   Get-Module
   ```

How it works...

The way to load SMO assemblies has changed between different versions of PowerShell. In PowerShell v1, loading assemblies can be done explicitly using the `Load()` or `LoadWithPartialName()` methods. `LoadWithPartialName()` accepts the partial name of the assembly, and loads from the application directory or the **Global Assembly Cache** (**GAC**):

```
[void][Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.
Smo")
```

Although `LoadWithPartialName()` is still supported and still remains a popular way of loading assemblies, this method should not be used because it will be deprecated in future versions.

`Load()` requires the fully qualified name of the assembly:

```
[void][Reflection.Assembly]::Load("Microsoft.SqlServer.Smo,
Version=9.0.242.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91")
```

In PowerShell V2, assemblies can be added by using `Add-Type`:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Smo"
```

In PowerShell V3, loading these assemblies one by one is no longer necessary as long as the `SQLPS` module is loaded:

```
Import-Module SQLPS
```

There may be cases where you will still want to load specific DLL versions if you are dealing with specific SQL Server versions. Or you may want to load only specific assemblies without loading the whole `SQLPS` module. In this case, the `Add-Type` command is still the viable method of bringing the assemblies in.

There's more...

When you import the `SQLPS` module, you might see an error about conflicting or unapproved verbs:

The names of some imported commands from the module `SQLPS` include unapproved verbs that might make them less discoverable. To find the commands with unapproved verbs, run the `Import-Module` command again with the `Verbose` parameter. For a list of approved verbs, type `Get-Verb`.

This means there are some cmdlets that do not conform to the PowerShell naming convention, but the module and its containing cmdlets are still all loaded into your host. To suppress this warning, import the module with the `–DisableNameChecking` parameter.

See also

- The *Installing SMO* recipe

Discovering SQL-related cmdlets and modules

In order to be effective at working with SQL Server and PowerShell, knowing how to explore and discover cmdlets, snap-ins, and modules is in order.

Getting ready

Log in to your SQL Server instance, and launch PowerShell ISE. If you prefer the console, you can also launch that instead.

How to do it...

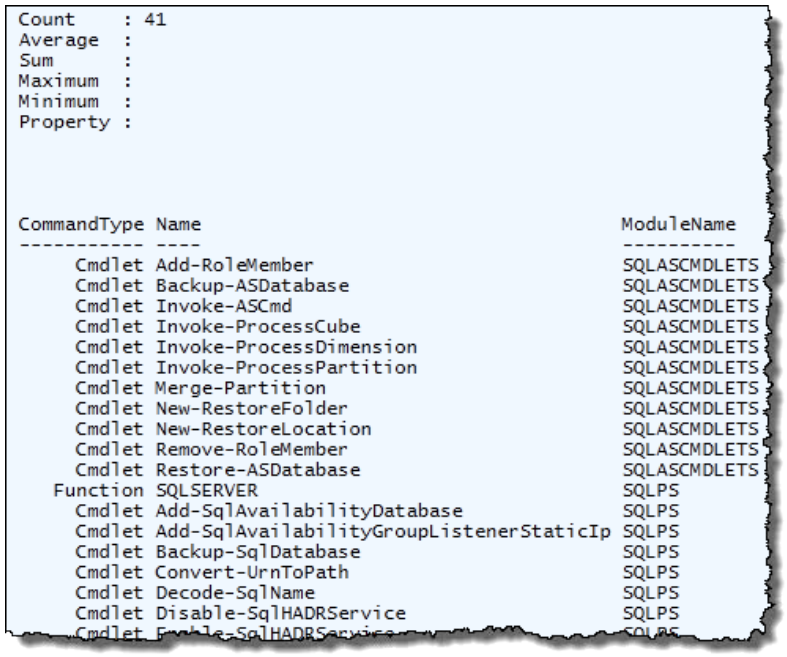
In this recipe we will list the SQL-Server related commands and cmdlets.

1. To discover SQL-related cmdlets, type the following in your PowerShell editor and run:

```
#how many commands from modules that
#have SQL in the name
Get-Command -Module "*SQL*" | Measure-Object

#list all the SQL-related commands
Get-Command -Module "*SQL*" |
Select CommandType, Name, ModuleName |
```

```
Sort -Property ModuleName, CommandType, Name |
Format-Table -AutoSize
```

After you execute the line, your output window should look similar to the following screenshot:



```
Count    : 41
Average  :
Sum      :
Maximum  :
Minimum  :
Property :



CommandType Name                                    ModuleName
----------- ----                                    ----------
     Cmdlet Add-RoleMember                          SQLASCMDLETS
     Cmdlet Backup-ASDatabase                       SQLASCMDLETS
     Cmdlet Invoke-ASCmd                            SQLASCMDLETS
     Cmdlet Invoke-ProcessCube                      SQLASCMDLETS
     Cmdlet Invoke-ProcessDimension                 SQLASCMDLETS
     Cmdlet Invoke-ProcessPartition                 SQLASCMDLETS
     Cmdlet Merge-Partition                         SQLASCMDLETS
     Cmdlet New-RestoreFolder                       SQLASCMDLETS
     Cmdlet New-RestoreLocation                     SQLASCMDLETS
     Cmdlet Remove-RoleMember                       SQLASCMDLETS
     Cmdlet Restore-ASDatabase                      SQLASCMDLETS
   Function SQLSERVER                               SQLPS
     Cmdlet Add-SqlAvailabilityDatabase             SQLPS
     Cmdlet Add-SqlAvailabilityGroupListenerStaticIp SQLPS
     Cmdlet Backup-SqlDatabase                      SQLPS
     Cmdlet Convert-UrnToPath                       SQLPS
     Cmdlet Decode-SqlName                          SQLPS
     Cmdlet Disable-SqlHADRService                  SQLPS
     Cmdlet Enable-SqlHADRService                   SQLPS
```

2. To see which of these modules are loaded, type the following in your editor and run:

   ```
   Get-Module -Name "*SQL*"
   ```

   If you have already used any of the cmdlets in the previous step, then you should see both **SQLPS** and **SQLASCMDLETS**. Otherwise, you will need to load these modules before you can use them.

3. To explicitly load these modules, type the following and run:

   ```
   Import-Module -Name "SQLPS"
   ```

   Note that **SQLASCMDLETS** will be loaded when you load **SQLPS**.

How it works...

At the core of PowerShell are cmdlets. A cmdlet (pronounced commandlet) can be a compiled, reusable .NET code, or an advanced function, or a workflow that typically performs a very specific task. All cmdlets follow the *verb-noun* naming notation.

PowerShell ships with many cmdlets and can be further extended if the shipped cmdlets are not sufficient for your purposes.

A legacy way of extending PowerShell is by registering additional snap-ins. A **snap-in** is a binary, or a DLL, that contains cmdlets. You can create your own by building your own .NET source, compiling, and registering the snap-in. You will always need to register snap-ins before you can use them. Snap-ins are a popular way of extending PowerShell.

The following table summarizes common tasks with snap-ins:

Task	Syntax
List loaded snap-ins	`Get-PSSnapin`
List installed snap-ins	`Get-PSSnapin -Registered`
Show commands in a snap-in	`Get-Command -Module "SnapinName"`
Load a specific snap-in	`Add-PSSnapin  "SnapinName"`

When starting, PowerShell V2, modules are available as the improved and preferred method of extending PowerShell.

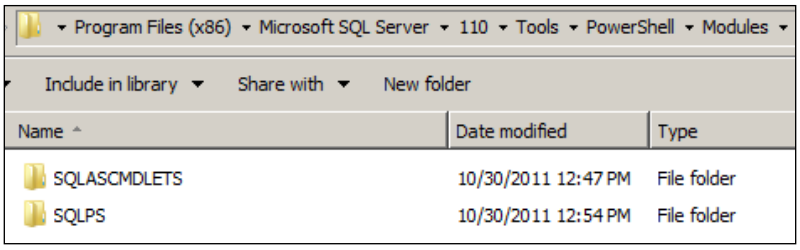
A module is a package that can contain cmdlets, providers, functions, variables, and aliases. In PowerShell V2, modules are not loaded by default, so required modules need to be explicitly imported.

Common tasks with modules are summarized in the following table:

Task	Syntax
List loaded modules	`Get-Module`
List installed modules	`Get-Module -ListAvailable`
Show commands in a module	`Get-Command -Module "ModuleName"`
Load a specific module	`Import-Module -Name "ModuleName"`

One of the improved features with PowerShell V3 is that it supports autoloading modules. You do not need to always explicitly load modules before using the contained cmdlets. Using the cmdlet in your script is enough to trigger PowerShell to load the module that contains it.

The SQL Server 2012 modules are located in the `PowerShell/Modules` folder of the `Install` directory:



There's more...

The following table shows the list of the `SQLPS` and `SQLASCMDLETS` cmdlets of this version:

CommandType Name	ModuleName
Cmdlet Add-RoleMember	SQLASCMDLETS
Cmdlet Backup-ASDatabase	SQLASCMDLETS
Cmdlet Invoke-ASCmd	SQLASCMDLETS
Cmdlet Invoke-ProcessCube	SQLASCMDLETS
Cmdlet Invoke-ProcessDimension	SQLASCMDLETS
Cmdlet Invoke-ProcessPartition	SQLASCMDLETS
Cmdlet Merge-Partition	SQLASCMDLETS
Cmdlet New-RestoreFolder	SQLASCMDLETS
Cmdlet New-RestoreLocation	SQLASCMDLETS
Cmdlet Remove-RoleMember	SQLASCMDLETS
Cmdlet Restore-ASDatabase	SQLASCMDLETS
Cmdlet Add-SqlAvailabilityDatabase	SQLPS
Cmdlet Add-SqlAvailabilityGroupListenerStaticIp	SQLPS
Cmdlet Backup-SqlDatabase	SQLPS
Cmdlet Convert-UrnToPath	SQLPS
Cmdlet Decode-SqlName	SQLPS
Cmdlet Disable-SqlHADRService	SQLPS
Cmdlet Enable-SqlHADRService	SQLPS
Cmdlet Encode-SqlName	SQLPS
Cmdlet Invoke-PolicyEvaluation	SQLPS
Cmdlet Invoke-Sqlcmd	SQLPS
Cmdlet Join-SqlAvailabilityGroup	SQLPS
Cmdlet New-SqlAvailabilityGroup	SQLPS
Cmdlet New-SqlAvailabilityGroupListener	SQLPS
Cmdlet New-SqlAvailabilityReplica	SQLPS
Cmdlet New-SqlHADREndpoint	SQLPS

CommandType Name	ModuleName
Cmdlet Remove-SqlAvailabilityDatabase	SQLPS
Cmdlet Remove-SqlAvailabilityGroup	SQLPS
Cmdlet Remove-SqlAvailabilityReplica	SQLPS
Cmdlet Restore-SqlDatabase	SQLPS
Cmdlet Resume-SqlAvailabilityDatabase	SQLPS
Cmdlet Set-SqlAvailabilityGroup	SQLPS
Cmdlet Set-SqlAvailabilityGroupListener	SQLPS
Cmdlet Set-SqlAvailabilityReplica	SQLPS
Cmdlet Set-SqlHADREndpoint	SQLPS
Cmdlet Suspend-SqlAvailabilityDatabase	SQLPS
Cmdlet Switch-SqlAvailabilityGroup	SQLPS
Cmdlet Test-SqlAvailabilityGroup	SQLPS
Cmdlet Test-SqlAvailabilityReplica	SQLPS
Test-SqlDatabaseReplicaState	SQLPS

To learn more about these cmdlets, use the `Get-Help` cmdlet. For example:

```
Get-Help "Invoke-Sqlcmd"
Get-Help "Invoke-Sqlcmd" -Detailed
Get-Help "Invoke-Sqlcmd" -Examples
Get-Help "Invoke-Sqlcmd" -Full
```

You can also check out the MSDN article on SQL Server database engine cmdlets:

`http://msdn.microsoft.com/en-us/library/cc281847.aspx`

When you load the `SQLPS` module, several assemblies are loaded into your host.

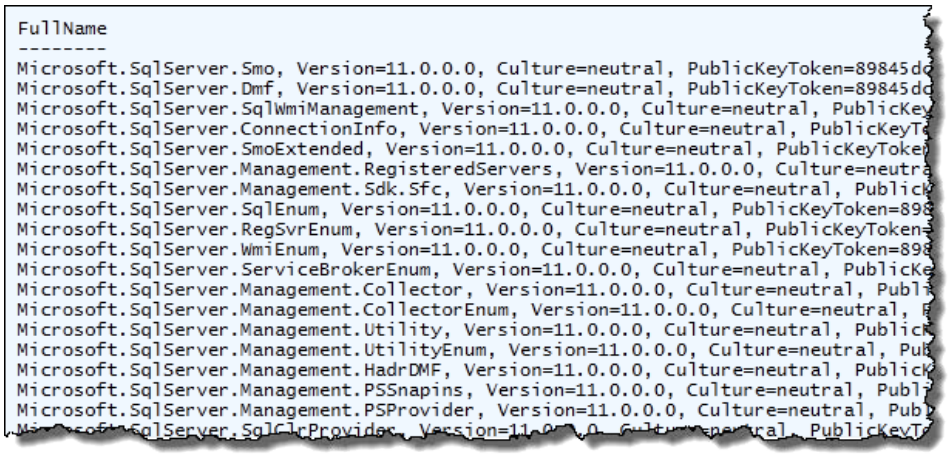
To get a list of SQL Server-related assemblies loaded with the `SQLPS` module, use the following script, which will work in both PowerShell V2 and V3:

```
Import-Module SQLPS –DisableNameChecking

[appdomain]::CurrentDomain.GetAssemblies() |
Where {$_.FullName -match "SqlServer" } |
Select FullName
```

If you want to run on a strictly V3 environment, you can take advantage of the simplified syntax:

```
Import-Module SQLPS –DisableNameChecking

[appdomain]::CurrentDomain.GetAssemblies() |
Where FullName -match "SqlServer" |
Select FullName
```

This will show you all the loaded assemblies, including their public key tokens:



More information on running PowerShell scripts

By default, PowerShell is running in restricted mode, in other words, it does not run scripts. To run our scripts from the book, we will set the execution policy to `RemoteSigned` as follows:

```
Set-ExecutionPolicy RemoteSigned
```

See the *Execution policy* section in *Appendix B*, *PowerShell Primer*, for further explanation of different execution policies.

If you save your PowerShell code in a file, you need to ensure it has a `.ps1` extension otherwise PowerShell will not run it. Ideally the filename you give your script does not have spaces. You can run this script from the PowerShell console simply by calling the name. For example if you have a script called `myscript.ps1` located in the `C:\` directory, this is how you would invoke it:

```
PS C:\> .\myscript.ps1
```

If the file or path to the file has spaces, then you will need to enclose the full path and file name in single or double quotes, and use the call (`&`) operator:

```
PS C:\>&'.\my script.ps1'
```

If you want to retain the variables and functions included in the script, in memory, thus making them available globally in your session, then you will need to dot source the script. To dot source is literally to prefix the filename, or the path to the file, with a dot and a space:

```
PS C:\> . .\myscript.ps1
PS C:\> . '.\my script.ps1'
```

More information on mixed assembly error

You may encounter an error when running some commands that are built using older .NET versions. Interestingly, you may see this while running your script in the PowerShell ISE, but not necessarily in the shell.

```
Invoke-Sqlcmd: Mixed mode assembly is built against version 'V2.0.50727'
of the runtime and cannot be loaded in the 4.0 runtime without additional
configuration information.
```

A few steps are required to solve this issue:

1. Open **Windows Explorer**.
2. Identify the Windows PowerShell ISE install folder path. You can find this out by going to **Start** | **All Programs** | **Accessories** | **Windows** | **PowerShell**, and then right-clicking on the **Windows PowerShell ISE** menu item and choosing **Properties**.

   For the 32-bit ISE, this is the default path:

   ```
   %windir%\sysWOW64\WindowsPowerShell\v1.0\PowerShell_ISE.exe
   ```

   For the 64-bit ISE, this is the default path:

   ```
   %windir%\system32\WindowsPowerShell\v1.0\PowerShell_ISE.exe
   ```

3. Go to the PowerShell ISE Install folder.
4. Create an empty file called `powershell_ise.exe.config`.
5. Add the following snippet to the content and save the file:

   ```
   <?xml version="1.0" encoding="utf-8" ?>
   <configuration>
   <startup useLegacyV2RuntimeActivationPolicy="true">
   <supportedRuntime version="v4.0" />
   </startup>

   <runtime>
   <generatePublisherEvidence enabled="false" />
   </runtime>
   </configuration>
   ```

6. Reopen PowerShell ISE and retry the command that failed.

Creating a SQL Server instance object

Most of what you will need to do in SQL Server will require a connection to an instance.

Getting ready

Open up your PowerShell console, the PowerShell ISE, or your favorite PowerShell editor.

You will need to note what your instance name is. If you have a default instance, you can use your machine name. If you have a named instance, the format will be `<machine name>\<instance name>`.

How to do it...

If you are connecting to your instance using Windows authentication, and using your current Windows login, follow these steps:

1. Import the `SQLPS` module:

```
#import SQLPS module
Import-Module SQLPS –DisableNameChecking
```

2. Store your instance name in a variable as follows:

```
#create a variable for your instance name
$instanceName = "KERRIGAN"
```

3. If you are connecting to your instance using Windows authentication using the account you are logged in as:

```
#create your server instance
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

   If you are connecting using SQL Authentication, you will need to know the username and password that you will use to authenticate. In this case, you will need to add the following code, which will set the connection to mixed mode and prompt for the username and password:

```
#set connection to mixed mode
$server.ConnectionContext.set_LoginSecure($false)
```

```
#set the login name
#of course we don't want to hardcode credentials here
#so we will prompt the user
#note password is passed as a SecureString type
$credentials = Get-Credential
#remove leading backslash in username
$login = $credentials.UserName -replace("\\", "")
$server.ConnectionContext.set_Login($login)
$server.ConnectionContext.set_SecurePassword($credentials.
Password)

#check connection string
$server.ConnectionContext.ConnectionString

Write-Verbose "Connected to $($server.Name)"
Write-Verbose "Logged in as $($server.ConnectionContext.
TrueLogin)"
```

How it works...

Before you can access or manipulate SQL Server programmatically, you will often need to create references to its objects. At the most basic is the server.

The server instance uses the type `Microsoft.SqlServer.Management.Smo.Server`. By default, connections to the server are made using trusted connections, meaning it uses the Windows account you're currently using when you log into the server. So all it needs is the instance name in its argument list:

```
#create your server instance
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

If, however, you need to connect using a SQL login, you will need to set the `ConnectionContext.LoginSecure` property of the SMO Server class setting to `false`:

```
#set connection to mixed mode
$server.ConnectionContext.set_LoginSecure($false)
```

You will also need to explicitly set the username and the password. The best way to accomplish this is to prompt the user for the credentials.

```
#prompt
$credentials = Get-Credential
```

The credential window will capture the login and password. The `Get-Credential` cmdlet returns the username with a leading backslash if the domain is not specified. In this case, we want to remove this leading backslash.

```
#remove leading backslash in username
$login = $credentials.UserName -replace("\\","")
```

Once we have the login, we can pass it to the `set_Login` method. The password is already a `SecureString` type, which is what the `set_SecurePassword` expects, so we can readily pass this to the method:

```
$server.ConnectionContext.set_Login($login)
$server.ConnectionContext.set_SecurePassword($credentials.Password)
```

Should you want to hardcode the username and just prompt for the password, you can also do this:

```
$login="belle"

#prompt
$credentials = Get-Credential –Credential $login
```

In the script, you will also notice we are using `Write-Verbose` instead of `Write-Host` to display our results. This is because we want to be able to control the output without needing to always go back to our script and remove all the `Write-Host` commands.

By default, the script will not display any output, that is, the `$VerbosePreference` special variable is set to `SilentlyContinue`. If you want to run the script in verbose mode, you simply need to add this line in the beginning of your script:

```
$VerbosePreference = "Continue"
```

When you are done, you just need to revert the value to `SilentlyContinue`:

```
$VerbosePreference = "SilentlyContinue"
```

See also

- The *Loading SMO assemblies* recipe
- The *Creating SQL Server instance using SMO* recipe

Exploring SMO server objects

SQL Management Objects (**SMO**) comes with a hierarchy of objects that are accessible programmatically. For example, when we create an SMO server variable, we can then access databases, logins, and database-level triggers. Once we get a handle of individual databases, we can then traverse the tables, stored procedures, and views that it contains. Since many tasks involve SMO objects, you will be at an advantage if you know how to discover and navigate these objects.

Getting ready

Open up your PowerShell console, the PowerShell ISE, or your favorite PowerShell editor.

You will also need to note what your instance name is. If you have a default instance, you can use your machine name. If you have a named instance, the format will be `<machine name>\<instance name>`

How to do it...

In this recipe, we will start exploring the hierarchy of objects with SMO.

1. Import the `SQLPS` module as follows:

```
Import-Module SQLPS -DisableNameChecking
```

2. Create a server instance as follows:

```
$instanceName = "KERRIGAN"

$server = New-Object `
        -TypeName Microsoft.SqlServer.Management.Smo.Server `
        -ArgumentList $instanceName
```

3. Get the SMO objects directly accessible from the `$server` object:

```
$server |
Get-Member -MemberType "Property" |
Where Definition -like "*Smo*"
```

4. Now let's check SMO objects under databases. Execute the following line:

```
$server.Databases |
Get-Member -MemberType "Property" |
Where Definition -like "*Smo*"
```

5. To check out the tables, you can type and execute:

```
$server.Databases["AdventureWorks2008R2"].Tables |
Get-Member -MemberType "Property" |
Where Definition -like "*Smo*"
```

How it works...

SMO contains a hierarchy of objects. At the very top there is a server object, which in turn contains objects such as `Databases`, `Configuration`, `SqlMail`, `LoginCollection`, and the like. These objects in turn contain other objects, for example, `Databases` is a collection that contains `Database` objects, and a `Database` in turn, contains `Tables` and so on.

See also

- The *Loading SMO assemblies* recipe
- The *Creating a SQL Server instance using SMO* recipe
- You can also check out the SMO object model diagram from MSDN:

  `http://msdn.microsoft.com/en-us/library/ms162209(SQL.110).aspx`

2

SQL Server and PowerShell Basic Tasks

In this chapter, we will cover:

- Listing SQL Server instances
- Discovering SQL Server services
- Starting/stopping SQL Server services
- Listing SQL Server configuration settings
- Changing SQL Server instance configurations
- Searching for database objects
- Creating a database
- Altering database properties
- Dropping a database
- Changing a database owner
- Creating a table
- Creating a view
- Creating a stored procedure
- Creating a trigger
- Creating an index
- Executing a query / SQL script
- Performing bulk export using `Invoke-Sqlcmd`
- Performing bulk export using `bcp`
- Performing bulk import using `BULK INSERT`
- Performing bulk import using `bcp`

Introduction

This chapter demonstrates scripts and snippets of code that accomplish some basic SQL Server tasks, using PowerShell. We will start with simple tasks, such as listing SQL Server instances and creating objects such as tables, indexes, stored procedures, and functions, to get you comfortable with working with SQL Server programmatically.

You will find that many of the recipes can be accomplished using PowerShell and **SQL Management Objects** (**SMO**). SMO is a library that exposes SQL Server classes, which allows for programmatic manipulation and automation of many database tasks. For some recipes, we will also explore alternative ways of accomplishing the same tasks, using different native PowerShell cmdlets.

SMO is explained in more detail in *Chapter 1*, *Getting Started with SQL Server and PowerShell*.

Even though we are exploring how to create some common database objects using PowerShell, I would like to note that PowerShell is not always the best tool for the task. There will be tasks that are best left accomplished using T-SQL. Even so, it is still good to know what is possible with PowerShell and how to do it, so that you know you have alternatives depending on your requirements or situation.

Development environment

The development environment used in the recipes has the following configurations:

Component	Syntax
Domain	`QUERYWORKS`
Machine name	`KERRIGAN`
Instances	`KERRIGAN` or `(local)` or `localhost`
	`SQL01`
Databases	`AdventureWorks2008R2`
Domain accounts	`QUERYWORKS\aterra`
	`QUERYWORKS\jraynor`
	`QUERYWORKS\mhorner`

Administrator

To simplify the exercises, run the PowerShell scripts as an administrator in your box. In addition, ensure this account has full access to the SQL Server instance on which you are working.

PowerShell ISE

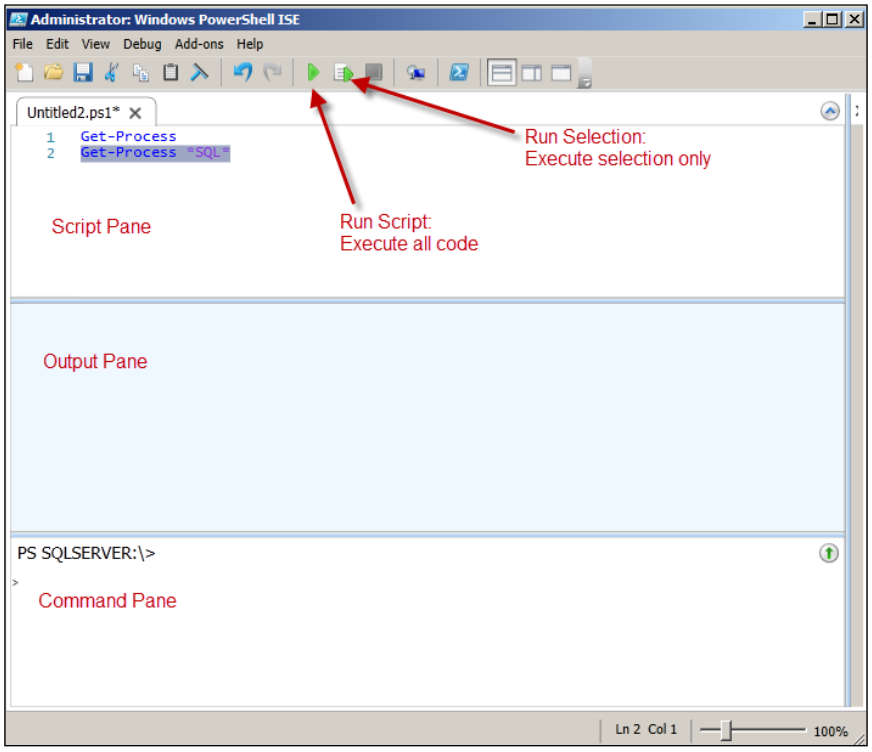
We will be using the PowerShell ISE for all the scripts in this task. These are some things you need to remember.

The **Script Pane** is where you will be typing in your PowerShell code. The **Output Pane** is where you will see the results.

The **Command Pane** is where you can type ad hoc commands, which get executed as soon as you press *Enter*.

For our recipes, we will be using the **Script Pane** to write and execute our scripts. Depending on the task, you may need to do one of the following:

- Click on the **Run Script** icon (green arrow) to run all code in the script
- Click on the **Run Selection** icon right beside it to run only highlighted code



Running scripts

If you prefer running the script from the PowerShell console rather than running the commands from the ISE, you can follow these steps:

6. Save the file with a `.ps1` extension.
7. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell**.
8. We want to be able to run locally created scripts. To do this, we need to change the `ExecutionPolicy` to `RemoteSigned`.
9. Set `ExecutionPolicy` to `RemoteSigned`.

See the *Execution Policy* section of the *Running PowerShell scripts* recipe in *Appendix B*, *PowerShell Primer,* for further explanation of different execution policies.

10. You can pick from the following options:

    - Change directory to where your script is stored and invoke your script in this way:

      ```
      PS C:\>.\SampleScript.ps1 param1 param2
      ```

    - Use the full qualified path to run the `.ps1` file:

      ```
      PS C:\>#if your path has no space
      PS C:\>C:\MyScripts\SampleScript.ps1 param1 param2

      PS C:\>#if your path has space
      PS C:\>& "C:\My Scripts\SampleScript.ps1" param1 param2
      ```

    - If you want to retain the functions and variables in your script throughout your session, you can dot source your file:

      ```
      PS C:\>. .\SampleScript.ps1 param1 param2
      PS C:\>. "C:\My Scripts\SampleScript.ps1" param1 param2
      ```

Listing SQL Server instances

In this recipe, we will list all SQL Server instances in the local network.

Getting ready

Log in to the server that has your SQL Server development instance, as an administrator.

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Let's use the `Start-Service` cmdlet to start SQLBrowser:

```
Import-Module SQLPS -DisableNameChecking

#sql browser must be installed and running
Start-Service "SQLBrowser"
```

3. Next, you need to create a `ManagedComputer` object to get access to instances. Type the following script and run it:

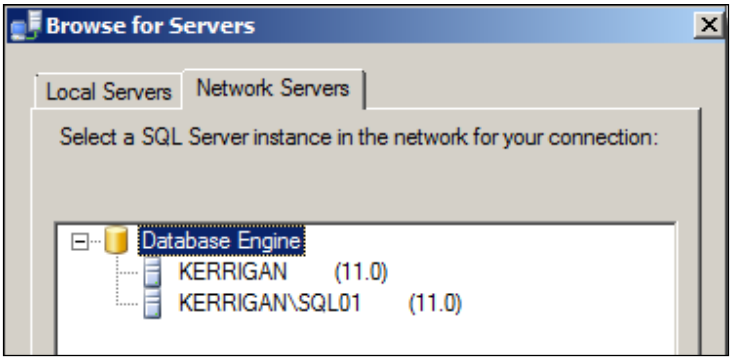
```
$instanceName = "KERRIGAN"
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.
Wmi.ManagedComputer' $instanceName

#list server instances
$managedComputer.ServerInstances
```

Your result should look similar to the one shown in the following screenshot:

```
ServerProtocols : {Np, Sm, Tcp}
Parent          : Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer
Urn             : ManagedComputer[@Name='KERRIGAN']/ServerInstance[@Name='MSSQLSERVER']
Name            : MSSQLSERVER
Properties      : {}
UserData        :
State           : Existing

ServerProtocols : {Np, Sm, Tcp}
Parent          : Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer
Urn             : ManagedComputer[@Name='KERRIGAN']/ServerInstance[@Name='SQL01']
Name            : SQL01
Properties      : {}
UserData        :
State           : Existing
```

Note that `$managedComputer.ServerInstances` gives you not only instance names, but also additional properties such as `ServerProtocols`, `Urn`, `State`, and so on.

4. Confirm that these are the same instances you see in **Management Studio**. Open up **Management Studio**.
5. Go to **Connect** | **Database Engine**.
6. In the **Server Name** drop-down, click on **Browse for More**.
7. Select the **Network Servers** tab, and check the instances listed. Your screen should look similar to this:



How it works...

All services in a Windows operating system are exposed and accessible using **Windows Management Instrumentation** (**WMI**). WMI is Microsoft's framework for listing, setting, and configuring any Microsoft-related resource. This framework follows **Web-based Enterprise Management** (**WBEM**). Distributed Management Task Force, Inc. defines WBEM as follows (`http://www.dmtf.org/standards/wbem`):

> *a set of management and internet standard technologies developed to unify the management of distributed computing environments. WBEM provides the ability for the industry to deliver a well-integrated set of standard-based management tools, facilitating the exchange of data across otherwise disparate technologies and platforms.*

In order to access SQL Server WMI-related objects, you can create a WMI `ManagedComputer` instance:

```
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.Wmi.
ManagedComputer' $instanceName
```

The `ManagedComputer` object has access to a `ServerInstance` property, which in turn lists all available instances in the local network. These instances, however, are only identifiable if the SQL Server Browser service is running.

SQL Server Browser is a Windows service that can provide information on installed instances in a box. You need to start this service if you want to list the SQL Server-related services.

There's more...

An alternative to using the `ManagedComputer` object is using the `System.Data.Sql.SQLSourceEnumerator` class to list all the SQL Server instances in the local network, thus:

```
[System.Data.Sql.SqlDataSourceEnumerator]::Instance.GetDataSources() |
Select ServerName, InstanceName, Version |
Format-Table -AutoSize
```

When you execute this, your result should look similar to the following screenshot:

```
ServerName InstanceName Version
---------- ------------ -------
KERRIGAN                11.0.1440.19
KERRIGAN   SQL01        11.0.1440.19
```

Yet another way to get a handle to the SQL Server WMI object is by using the `Get-WmiObject` cmdlet. This will not, however, expose exactly the same properties exposed by the `Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer` object.

To do this, you will need to discover first what namespace is available in your environment, thus:

```
$hostname = "KERRIGAN"

$namespace = Get-WMIObject -ComputerName $hostName -NameSpace root\
Microsoft\SQLServer -Class "__NAMESPACE" |
            Where Name -Like "ComputerManagement*"
```

If you are using PowerShell V2, you will have to change the `Where` cmdlet usage to use the curly braces ({}) and the `$_` variable, thus:

```
Where {$_.Name -Like "ComputerManagement*" }
```

For SQL Server 2012, this value is:

```
ROOT\Microsoft\SQLServer\ComputerManagement11
```

Once you have the namespace, you can use this value with `Get-WmiObject` to retrieve the instances. One property we can use to filter is `SqlServiceType`.

According to MSDN (`http://msdn.microsoft.com/en-us/library/ms179591.aspx`), the following are the values of `SqlServiceType`:

SqlServiceType	Description
1	SQL Server service
2	SQL Server Agent service
3	Full-text Search Engine service
4	Integration Services service
5	Analysis Services service
6	Reporting Services service
7	SQL Server Browser service

Thus, to retrieve the SQL Server instances, you need to filter for SQL Server service, or `SQLServiceType = 1`.

```
Get-WmiObject -ComputerName $hostname `
-Namespace "$($namespace.__NAMESPACE)\$($namespace.Name)" `
-Class SqlService |
Where SQLServiceType -eq 1 |
Select ServiceName, DisplayName, SQLServiceType |
Format-Table –AutoSize
```

If you are using PowerShell V2, you will have to change the `Where` cmdlet usage to use the curly braces ({ }) and the `$_` variable:

```
Where {$_.SQLServiceType -Like –eq 1 }
```

Your result should look similar to the following screenshot:

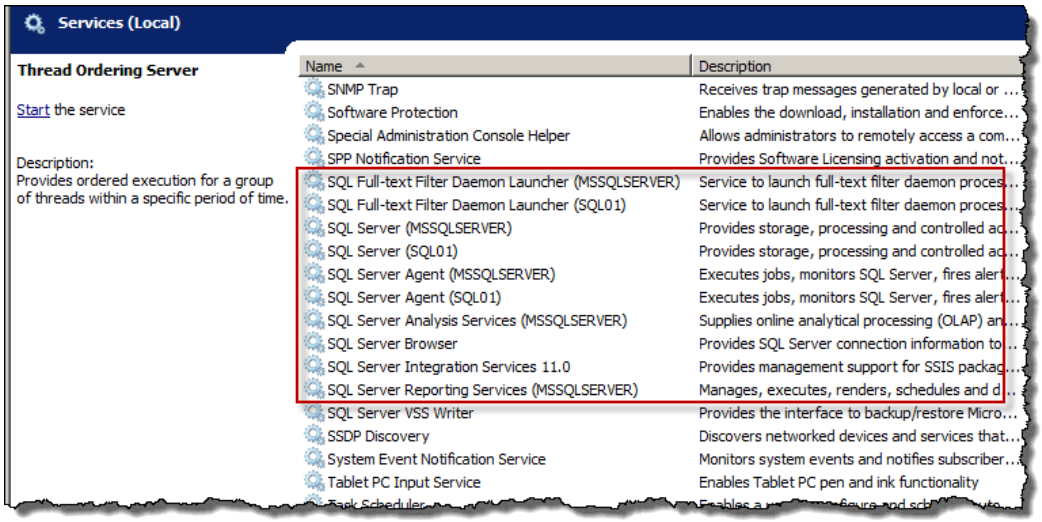
```
ServiceName DisplayName              SQLServiceType
----------- -----------              --------------
MSSQL$SQL01 SQL Server (SQL01)                    1
MSSQLSERVER SQL Server (MSSQLSERVER)              1
```

Discovering SQL Server services

In this recipe, we enumerate all SQL Server services and list their status.

Getting ready

Check which SQL Server services are installed in your instance. Go to **Start** | **Run** and type `services.msc`. You should see a screen similar to this:



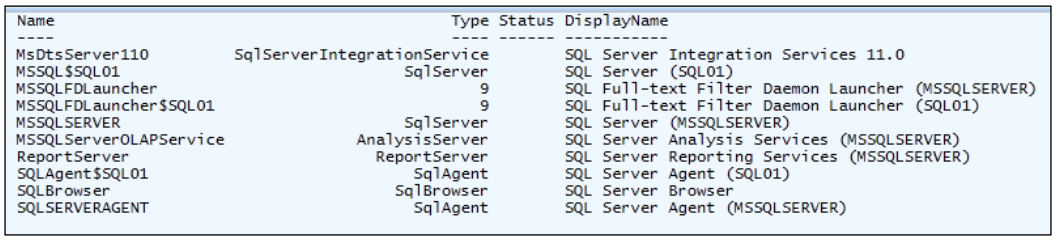
How to do it...

Let's assume you are running this script on the server box.

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Add the following code and execute it:

```
Import-Module SQLPS

#replace KERRIGAN with your instance  name
$instanceName = "KERRIGAN"
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.
Wmi.ManagedComputer' $instanceName

#list services
$managedComputer.Services |
Select Name, Type, Status, DisplayName |
Format-Table -AutoSize
```

Your result will look similar to the one shown in the following screenshot:



```
Name                                          Type Status DisplayName
----                                          ---- ------ -----------
MsDtsServer110         SqlServerIntegrationService        SQL Server Integration Services 11.0
MSSQL$SQL01                          SqlServer        SQL Server (SQL01)
MSSQLFDLauncher                              9        SQL Full-text Filter Daemon Launcher (MSSQLSERVER)
MSSQLFDLauncher$SQL01                        9        SQL Full-text Filter Daemon Launcher (SQL01)
MSSQLSERVER                          SqlServer        SQL Server (MSSQLSERVER)
MSSQLServerOLAPService            AnalysisServer        SQL Server Analysis Services (MSSQLSERVER)
ReportServer                        ReportServer        SQL Server Reporting Services (MSSQLSERVER)
SQLAgent$SQL01                           SqlAgent        SQL Server Agent (SQL01)
SQLBrowser                             SqlBrowser        SQL Server Browser
SQLSERVERAGENT                           SqlAgent        SQL Server Agent (MSSQLSERVER)
```

Items listed on your screen will vary depending on the features installed and running in your instance.

3. Confirm that these are the services that exist in your server. Check your services window.

How it works...

Services that are installed on a system can be queried using WMI. Specific services for SQL Server are exposed through SMO's WMI `ManagedComputer` object. Some of the exposed properties include:

- `ClientProtocols`
- `ConnectionSettings`
- `ServerAliases`
- `ServerInstances`
- `Services`

There's more...

An alternative way to get SQL Server-related services is by using `Get-WMIObject`. We will need to pass in the hostname, as well as SQL Server WMI provider for the Computer Management namespace. For SQL Server 2012, this value is:

```
ROOT\Microsoft\SQLServer\ComputerManagement11
```

The script to retrieve the services is provided in the following code. Note that we are dynamically composing the WMI namespace here.

```
$hostName = "KERRIGAN"

$namespace = Get-WMIObject -ComputerName $hostName -NameSpace root\
Microsoft\SQLServer -Class "__NAMESPACE" |
```

```
                Where Name -Like "ComputerManagement*"
Get-WmiObject -ComputerName $hostname -Namespace "$($namespace.__
NAMESPACE)\$($namespace.Name)" -Class SqlService |
Select ServiceName
```

Yet another alternative but *less accurate* way of listing possible SQL Server-related services is the following snippet of code:

```
#alterative - but less accurate
Get-Service *SQL*
```

It uses the `Get-Service` cmdlet and filters based on the service name. It is less accurate because this cmdlet grabs all processes that have SQL in the name but may not necessarily be SQL Server-related. For example, if you have MySQL installed, that will get picked up as a process. Conversely, this cmdlet will not pick up SQL Server-related services that do not have SQL in the name, such as `ReportServer`.

See also

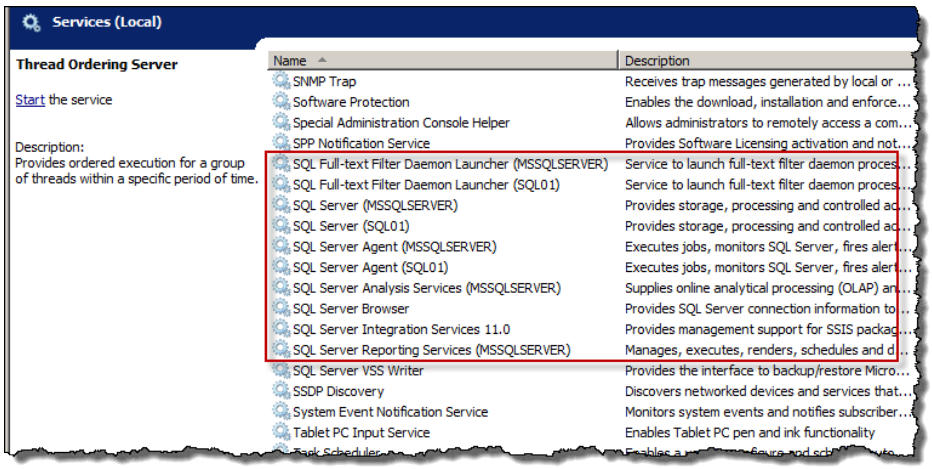
- The *Listing SQL Server instances* recipe

Starting/stopping SQL Server services

This recipe describes how to start and/or stop SQL Server services.

Getting ready

Check which SQL services are installed in your machine. Go to **Start** | **Run** and type `Services.msc`. You should see a screen similar to this:



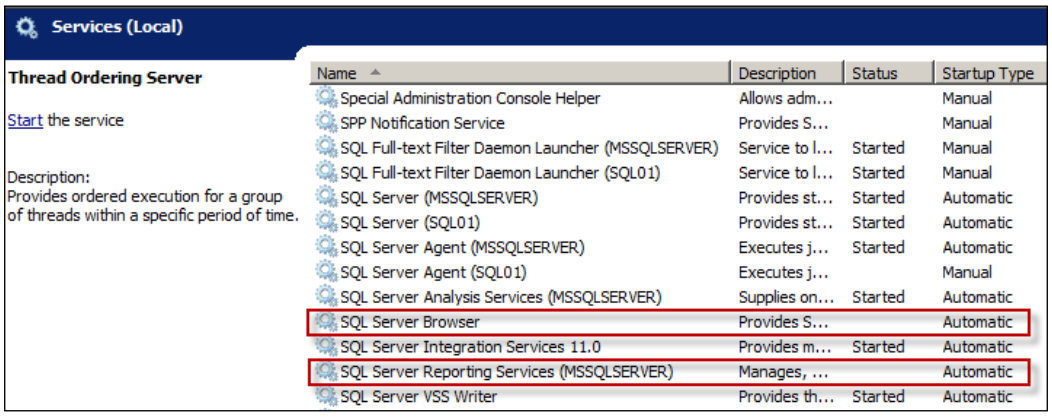
How to do it...

Let's look at the steps to toggle states for your SQL Server services:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Add the following code. Note that this code will work in both PowerShell V2 and V3:

```
$Verbosepreference = "Continue"
$services = @("SQLBrowser", "ReportServer")
$hostName = "KERRIGAN"

$services | ForEach-Object {
   $service = Get-Service -Name $_
   if($service.Status -eq "Stopped")
   {
      Write-Verbose "Starting $($service.Name) ...."
      Start-Service -Name $service.Name
   }
   else
   {
      Write-Verbose "Stopping $($service.Name) ...."
      Stop-Service -Name $service.Name
   }
}
$VerbosePreference = "SilentlyContinue"
```

3. Execute and confirm the service status changed accordingly. Go to **Start** | **Run** and type `Services.msc`.



For example, in our previous sample, both **SQLBrowser** and **ReportServer** were initially running. Once the script was executed, both services stopped.

How it works...

In this recipe, we picked two services—**SQLBrowser** and **ReportServer**—that we want to manipulate and saved them into an array:

```
$services = @("SQLBrowser","ReportServer")
```

We then pipe the array contents to a `Foreach-Object` cmdlet, so we can determine what action to perform for each service. For our purposes, if the service is stopped, we want to start it. Otherwise, we stop it. Note that this code will work in both PowerShell V2 and V3:

```
$services | ForEach-Object {
   $service = Get-Service -Name $_
   if($service.Status -eq "Stopped")
   {
      Write-Verbose "Starting $($service.Name) ...."
      Start-Service -Name $service.Name
   }
   else
   {
      Write-Verbose "Stopping $($service.Name) ...."
      Stop-Service -Name $service.Name
   }
}
```

You may also want to determine dependent services, or services that rely on a particular service. You may want to consider synchronizing the starting/stopping of these services with the main service they depend on.

To identify dependent services, you can use the `DependentServices` property of the `System.ServiceProcess.ServiceController` class:

```
$services | ForEach-Object {
   $service = Get-Service -Name $_
   Write-Verbose "Services Dependent on $($service.Name)"
   $service.DependentServices | Select Name
}
```

The following list shows the properties and methods of the `System.ServiceProcess.ServiceController` class, which is generated from the `Get-Service` cmdlet:

```
Name                       MemberType
----                       ----------
Name                       AliasProperty
RequiredServices           AliasProperty
Disposed                   Event
Close                      Method
Continue                   Method
CreateObjRef               Method
Dispose                    Method
Equals                     Method
ExecuteCommand             Method
GetHashCode                Method
GetLifetimeService         Method
GetType                    Method
InitializeLifetimeService  Method
Pause                      Method
Refresh                    Method
Start                      Method
Stop                       Method
WaitForStatus              Method
CanPauseAndContinue        Property
CanShutdown                Property
CanStop                    Property
Container                  Property
DependentServices          Property
DisplayName                Property
MachineName                Property
ServiceHandle              Property
ServiceName                Property
ServicesDependedOn         Property
ServiceType                Property
Site                       Property
Status                     Property
ToString                   ScriptMethod
```

An alternative way of working with SQL Server services is by using the `Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer` class. Note that the following code will work in both PowerShell V2 and V3:

```
Import-Module SQLPS -DisableNameChecking

#list services you want to start/stop here
$services = @("SQLBrowser", "ReportServer")
$instanceName = "KERRIGAN"
$managedComputer = New-Object 'Microsoft.SqlServer.Management.Smo.Wmi.
ManagedComputer' $instanceName

#go through each service and toggle the state
$services | ForEach-Object {
   $service = $managedComputer.Services[$_]
   switch($service.ServiceState)
   {
      "Running"
```

```
    {
            Write-Verbose "Stopping $($service.Name)"
            $service.Stop()
        }
        "Stopped"
        {
            Write-Verbose "Starting $($service.Name)"
            $service.Start()
        }
    }
}
```

When using the `Smo.Wmi.ManagedComputer` object, you can simply use the `Stop` method provided with the class and the `Start` method to stop and start the service respectively.

The following list shows the properties and methods available with the `Smo.Wmi.ManagedComputer` class:

```
   TypeName: Microsoft.SqlServer.Management.Smo.Wmi.Service

Name                      MemberType Definition
----                      ---------- ----------
ManagementStateChange     Event      System.Management.CompletedEvent
Alter                     Method     System.Void Alter()
ChangeHadrServiceSetting  Method     System.Void ChangeHadrServiceSet
ChangePassword            Method     System.Void ChangePassword(stri
Equals                    Method     bool Equals(System.Object obj)
GetHashCode               Method     int GetHashCode()
GetType                   Method     type GetType()
Initialize                Method     bool Initialize()
Pause                     Method     System.Void Pause()
Refresh                   Method     System.Void Refresh()
Resume                    Method     System.Void Resume()
SetServiceAccount         Method     System.Void SetServiceAccount(st
Start                     Method     System.Void Start()
Stop                      Method     System.Void Stop()
ToString                  Method     string ToString()
Validate                  Method     Microsoft.SqlServer.Management.S
AcceptsPause              Property   System.Boolean AcceptsPause {get
AcceptsStop               Property   System.Boolean AcceptsStop {get;
AdvancedProperties        Property   Microsoft.SqlServer.Management.S
Dependencies              Property   System.Collections.Specialized.S
Description               Property   System.String Description {get;}
DisplayName               Property   System.String DisplayName {get;}
ErrorControl              Property   Microsoft.SqlServer.Management.S
ExitCode                  Property   System.Int32 ExitCode {get;}
IsHadrEnabled             Property   System.Boolean IsHadrEnabled {ge
Name                      Property   System.String Name {get;set;}
Parent                    Property   Microsoft.SqlServer.Management.S
PathName                  Property   System.String PathName {get;}
ProcessId                 Property   System.Int32 ProcessId {get;}
Properties                Property   Microsoft.SqlServer.Management.S
ServiceAccount            Property   System.String ServiceAccount {ge
ServiceState              Property   Microsoft.SqlServer.Management.S
StartMode                 Property   Microsoft.SqlServer.Management.S
StartupParameters         Property   System.String StartupParameters
State                     Property   Microsoft.SqlServer.Management.S
Type                      Property   Microsoft.SqlServer.Management.S
Urn                       Property   Microsoft.SqlServer.Management.S
UserData                  Property   System.Object UserData {get;set;
```

There's more...

To explore available cmdlets that can help manage and maintain services, use the following command:

```
Get-Command -Name *Service* -CommandType Cmdlet -ModuleName
*PowerShell*
```

This will enumerate all cmdlets that have "Service" in the name:

```
CommandType     Name
-----------     ----
Cmdlet          Get-Service
Cmdlet          New-Service
Cmdlet          New-WebServiceProxy
Cmdlet          Restart-Service
Cmdlet          Resume-Service
Cmdlet          Set-Service
Cmdlet          Start-Service
Cmdlet          Stop-Service
Cmdlet          Suspend-Service
```

All of these cmdlets relate to Windows services, with the exception of `New-WebServiceProxy`, which is described in MSDN as a cmdlet that *creates a Web service proxy object that lets you use and manage the Web service in Windows PowerShell*.

Here is a brief comparison between these service-oriented cmdlets and the methods available for the object of `Microsoft.SqlServer.Management.Smo.Wmi.ManagedComputer` service, as discussed in the recipe:

Service Methods	Service-related cmdlets
`Start()`	`Start-Service`
`Stop()`	`Stop-Service`
`Continue()`	`Resume-Service`
`Pause()`	`Suspend-Service`
`Refresh()`	
	`Restart-Service`

Note that there isn't necessarily a one-to-one mapping between the methods of the `Service` class and the service cmdlets. For example, there is a `Restart-Service` cmdlet, but there isn't a `Restart` method.

This should not raise alarm bells, though. Although it may seem that some methods or cmdlets may be missing, it is important to note that PowerShell is a rich scripting platform and language. In addition to its own cmdlets, it leverages the whole .NET platform. Whatever you can do in the .NET platform, you most likely can do using PowerShell. Even if you think something is not doable when you look at a specific class or object, there is most likely a cmdlet somewhere that can perform that same task, or vice versa. If you still cannot find your ideal solution, you can create your own—be it a class, a module, a cmdlet, or a function.

See also

- The *Discovering SQL Server services* recipe

Listing SQL Server configuration settings

This recipe walks through how to list SQL Server configurable and non-configurable instance settings using PowerShell.

How to do it...

1. Open the **PowerShell ISE**. Go to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

   To explore what members and methods are included in the SMO server, use the following code snippet in PowerShell V3:

```
#Explore: get all properties available for a server object
#http://msdn.microsoft.com/en-us/library/ms212724.aspx
$server | Get-Member | Where MemberType -eq "Property"
```

   In PowerShell V2, you will need to slightly modify your syntax:

```
$server | Get-Member | Where {$_.MemberType -eq "Property"}
```

```
#The Information class lists nonconfigrable instance settings,
#like BuildNumber, OSVersion, ProductLevel etc
#Also includes settings specified during install
$server.Information.Properties |
Select Name, Value |
Format-Table -AutoSize
```

```
Name                                                                                Value
----                                                                                -----
BuildNumber                                                                          1440
Edition                                                    Enterprise Evaluation Edition (64-bit)
ErrorLogPath                  C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\Log
HasNullSaPassword
IsCaseSensitive                                                                     False
IsFullTextInstalled                                                                  True
Language                                                              English (United States)
MasterDBLogPath             C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA
MasterDBPath                C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA
MaxPrecision                                                                        38
NetName                                                                        KERRIGAN
OSVersion                                                                   6.1 (7601)
PhysicalMemory                                                                    2047
Platform                                                                        NT x64
Processors                                                                           1
Product                                                                 Microsoft SQL Server
RootDirectory                    C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL
VersionMajor                                                                        11
VersionMinor                                                                         0
VersionString                                                              11.0.1440.19
Collation                                                    SQL_Latin1_General_CP1_CI_AS
EngineEdition                                                                        3
IsClustered                                                                     False
IsSingleUser                                                                    False
ProductLevel                                                                      CTP
BuildClrVersionString                                                        v4.0.30319
CollationID                                                                  872468488
ComparisonStyle                                                                 196609
ComputerNamePhysicalNetBIOS                                                    KERRIGAN
```

3. Next, let's look at the `Settings` class:

```
#The Settings lists some instance level configurable settings,
#like LoginMode, BackupDirectory etc
$server.Settings.Properties |
Select Name, Value |
Format-Table -AutoSize
```

```
Name                                                                                                 Value
----                                                                                                 -----
AuditLevel                                                                                         Failure
BackupDirectory  C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\Backup
DefaultFile
DefaultLog
LoginMode                                                                                          Mixed
MailProfile
NumberOfLogFiles                                                                                      -1
PerfMonMode                                                                                         None
TapeLoadWaitTime                                                                                      -1
```

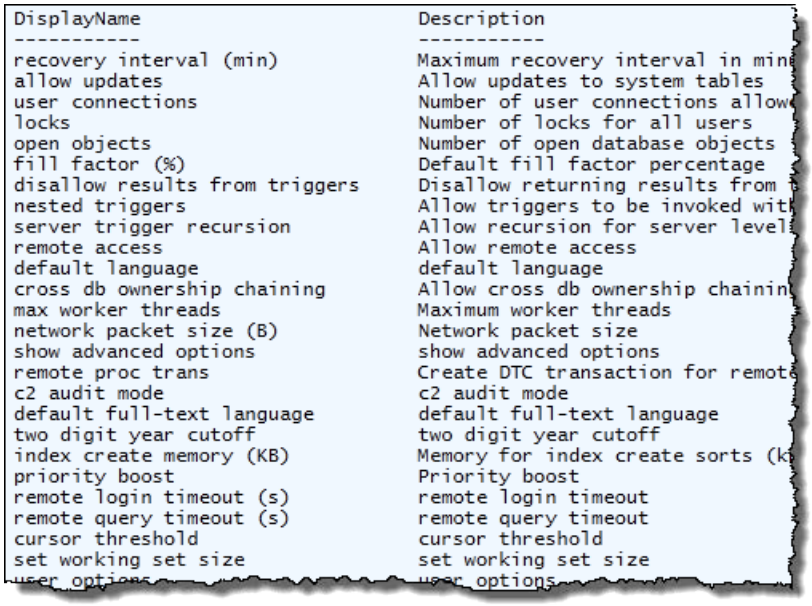
4. The `UserOptions` class lists user-specific options:

```
#The UserOptions include options that can be set for user
#connections, for example
#AnsiPadding, AnsiNulls, NoCount, QuotedIdentifier
$server.UserOptions.Properties |
Select Name, Value |
Format-Table -AutoSize
```

```
Name                          Value
----                          -----
AbortOnArithmeticErrors       False
AbortTransactionOnError       False
AnsiNullDefaultOff            False
AnsiNullDefaultOn             False
AnsiNulls                     False
AnsiPadding                   False
AnsiWarnings                  False
ConcatenateNullYieldsNull     False
CursorCloseOnCommit           False
DisableDefaultConstraintCheck False
IgnoreArithmeticErrors        False
ImplicitTransactions          False
NoCount                       False
NumericRoundAbort             False
QuotedIdentifier              False
```

5. The `Configuration` class contains instance-specific settings, similar to what you will see when you run `sp_configure`.

```
#The Configuration class contains instance specific settings,
#like AgentXPs, clr enabled, xp_cmdshell
#You will normally see this when you run
#the stored procedure sp_configure
$server.Configuration.Properties |
Select DisplayName, Description, RunValue, ConfigValue |
Format-Table -AutoSize
```



```
DisplayName                    Description
-----------                    -----------
recovery interval (min)        Maximum recovery interval in min
allow updates                  Allow updates to system tables
user connections               Number of user connections allow
locks                          Number of locks for all users
open objects                   Number of open database objects
fill factor (%)                Default fill factor percentage
disallow results from triggers Disallow returning results from
nested triggers                Allow triggers to be invoked wit
server trigger recursion       Allow recursion for server level
remote access                  Allow remote access
default language               default language
cross db ownership chaining    Allow cross db ownership chainin
max worker threads             Maximum worker threads
network packet size (B)        Network packet size
show advanced options          show advanced options
remote proc trans              Create DTC transaction for remot
c2 audit mode                  c2 audit mode
default full-text language     default full-text language
two digit year cutoff          two digit year cutoff
index create memory (KB)       Memory for index create sorts (k
priority boost                 Priority boost
remote login timeout (s)       remote login timeout
remote query timeout (s)       remote query timeout
cursor threshold               cursor threshold
set working set size           set working set size
user options                   user options
```

How it works...

Most SQL Server settings and configurations are exposed using SMO or WMI, which allows for these values to be programmatically retrieved.

At the core of accessing configuration details is the SMO Server class. This class exposes a SQL Server instance's properties, some of which are configurable, while some are not.

To create an SMO Server class, you will need to know your instance name and pass it as an argument:

```
#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

The following are the four main properties that store settings/configurations that we looked at in this recipe:

Server property	Description
`Information`	Includes non-configurable instance settings, such as `BuildNumber`, `Edition`, `OSVersion`, and `ProductLevel` It also includes settings specified during install, for example `Collation`, `MasterDBPath`, and `MasterDBLogPath`
`Settings`	Lists some instance-level configurable settings, such as `LoginMode` and `BackupDirectory`
`UserOptions`	Contain options that can be set for user connections, such as `AnsiWarnings`, `AnsiNulls`, `AnsiPadding`, and `NoCount`
`Configuration`	Instance-specific settings, such as `AgentXPs`, `remote access`, `clr enabled`, and `xp_cmdshell`, which you will normally see and set when you use the `sp_configure` system stored procedure

See also

- Check out MSDN for complete documentation on SMO classes:
  `http://msdn.microsoft.com/en-us/library/ms212724.aspx`

Changing SQL Server instance configurations

This recipe walks through how to change instance configuration settings using PowerShell.

Getting ready

For this recipe, we will:

- Change `FillFactor` to 60 percent
- Enable SQL Server Agent
- Set minimum server memory to 500 MB
- Change authentication method to `Mixed`

How to do it...

Let's change some SQL Server settings using PowerShell:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

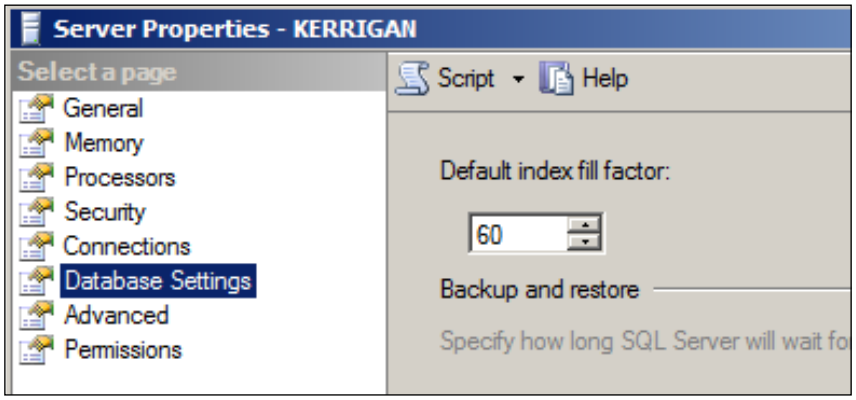
```
<#
run value vs config value
config_value," is what the setting has been set to (but may or
may not be what SQL Server is actually running now. Some settings
don't go into effect until SQL Server has been restarted, or
until the RECONFIGURE WITH OVERRIDE option has been run, as
appropriate.) And the last column, "run_value," is the value of
the setting currently in effect.
#>

#change FillFactor
$server.Configuration.FillFactor.ConfigValue = 60

#enable SQL Server Agent extended stored procedures
$server.Configuration.AgentXPsEnabled.ConfigValue = 1

#change minimum server memory to 500MB; MB is default
$server.Configuration.MinServerMemory.ConfigValue = 500

$server.Configuration.Alter()

#confirm changes
$server.Configuration.Properties |
Select DisplayName, ConfigValue |
Format-Table -AutoSize
```

```
#change authentication mode
$server.Settings.LoginMode = [Microsoft.SqlServer.Management.Smo.
ServerLoginMode]::Mixed
$server.Alter()

#confirm changes
$server.settings.LoginMode
```

4. Confirm the changes.

To confirm **fill factor**:

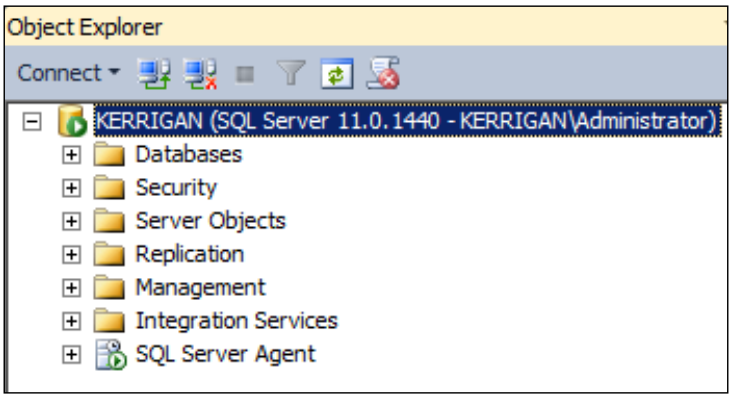
1. Go to **Management Studio**.
2. Connect to your instance.
3. Right-click on your instance and select **Properties**.
4. Go to **Database Settings**, and check whether your **fill factor** value has changed.



A side effect of enabling SQL Server Agent extended stored procedures is enabling SQL Server Agent. To confirm SQL Server Agent has been enabled:

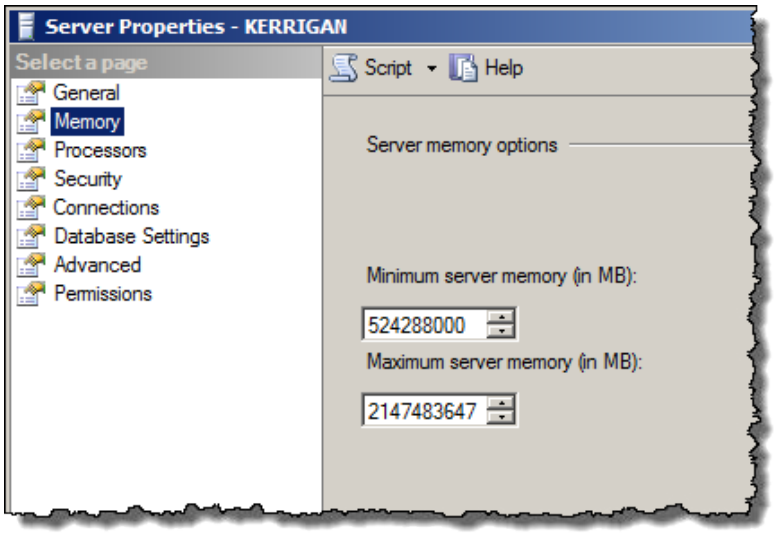
1. Go to **Management Studio**.
2. Connect to your instance.

3. Visually check whether **SQL Server Agent** for the instance you modified is now running.



To confirm **Minimum server memory**:

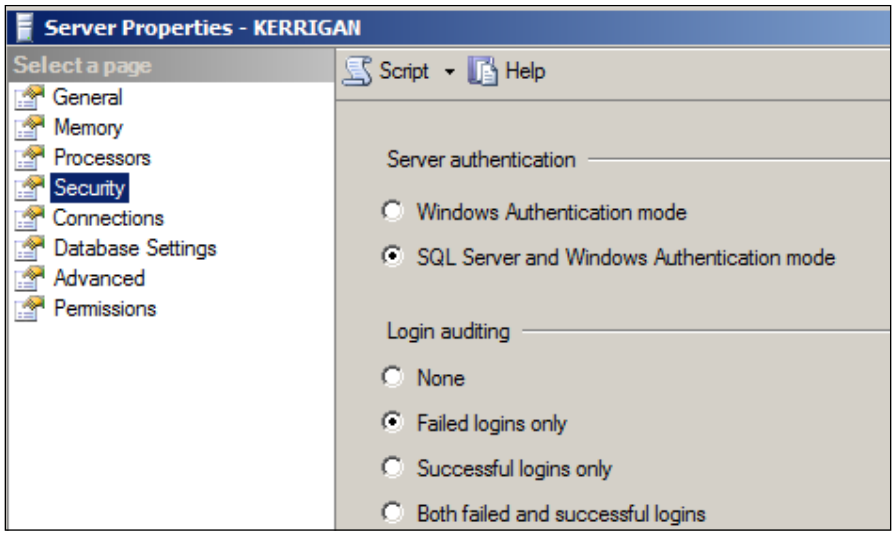
1. Go to **Management Studio.**
2. Right-click on your instance and select **Properties**.
3. Go to **Memory** and check that the value has changed to what you set it to.



To confirm authentication mode:

1. Go to **Management Studio**.
2. Connect to your instance.

3. Right-click on your instance and select **Properties**.
4. Go to **Security** and check that the instance is now **SQL Server and Windows Authentication mode**.



How it works...

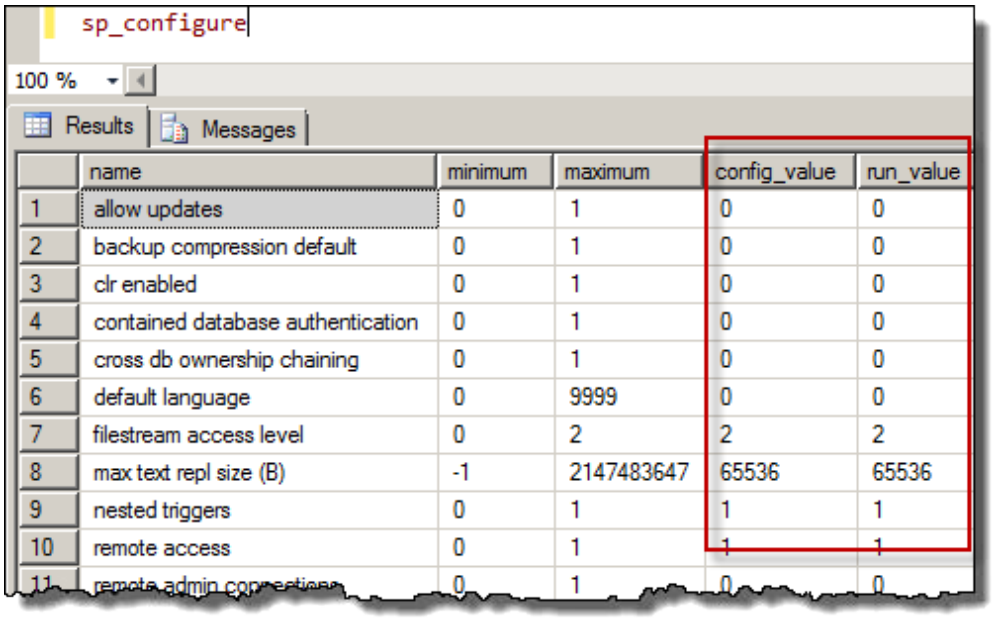
Depending on what server properties you need to change, you may need to determine which of the following classes you may need to access: `Settings`, `UserOptions`, or `Configuration`.

Once you have determined which class and property you want to change, you can change the values and invoke the `Alter` method:

```
#to make Configuration changes permanent
$server.Configuration.Alter()

#to make Settings changes permanent
$server.Alter()
```

There's more...

When you run **sp_configure**, you will see a result that shows both **run_value** and **config_value** as follows:



	name	minimum	maximum	config_value	run_value
1	allow updates	0	1	0	0
2	backup compression default	0	1	0	0
3	clr enabled	0	1	0	0
4	contained database authentication	0	1	0	0
5	cross db ownership chaining	0	1	0	0
6	default language	0	9999	0	0
7	filestream access level	0	2	2	2
8	max text repl size (B)	-1	2147483647	65536	65536
9	nested triggers	0	1	1	1
10	remote access	0	1	1	1
11	remote admin connections	0	1	0	0

There is often confusion between **run_value** and **config_value**. **config_value** is what value the setting is set to. **run_value** is what SQL Server is currently using. Sometimes, a new value may be set (**config_value**), but it isn't used by SQL Server until the instance is restarted.

See also

- The *Listing SQL Server configuration settings* recipe

Searching for database objects

In this recipe, we will search for database objects based on a search string by using PowerShell.

Getting ready

We will use `AdventureWorks2008R2`, in this exercise, and will look for SQL Server objects with the word "Product" in their names.

To get an idea of what are expecting to retrieve, run the following script in SQL Server Management Studio:

```
USE AdventureWorks2008R2
GO
SELECT
 *
FROM
 sys.objects
WHERE
 name LIKE '%Product%'
 -- filter table level objects only
 AND [type] NOT IN ('C', 'D', 'PK', 'F')
ORDER BY
 [type]
```

This will get you 23 results. Remember this number.

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it. Note that the following script will work only with PowerShell V3, because of the simplified `Where` cmdlet usage. If you want to use this in PowerShell V2, replace the `Where` syntax with the V2 variation.

```
$databaseName = "AdventureWorks2008R2"
$db = $server.Databases[$databaseName]

#what keyword are we looking for?
$searchString = "Product"

#create empty array, we will store results here
$results = @()
```

```
#now we will loop through all database SMO
#properties and look of objects that match
#the search string
#note we are explicitly excluding Federations, because
#this throws an error
$db |
Get-Member -MemberType Property |
Where Definition -Like "*Smo*" |
Where Definition -NotLike "*Federation*" |
ForEach-Object {
   $type = $_.Name
   $db.$type |
   Where Name -Like "*$searchstring*" |
   ForEach-Object {
       $result = New-Object -Type PSObject -Prop @{
                   "ObjectType"=$type.Replace("Microsoft.
SqlServer.Management.Smo.", "")
                   "ObjectName"=$_.Name
                 }
       $results += $result
    }

}

#display results
$results

#export results to csv file
$file = "C:\Temp\SearchResults.csv"
$results | Export-Csv -Path $file -NoTypeInformation

#display file contents
notepad $file
```

Your results will look like this:

```
ObjectType           ObjectName
----------           ----------
Schemas              Production
StoredProcedures     uspGetWhereUsedProductID
Tables               Product
Tables               ProductCategory
Tables               ProductCostHistory
Tables               ProductDescription
Tables               ProductDocument
Tables               ProductInventory
Tables               ProductListPriceHistory
Tables               ProductModel
Tables               ProductModelIllustration
Tables               ProductModelProductDescriptionCulture
Tables               ProductPhoto
Tables               ProductProductPhoto
Tables               ProductReview
Tables               ProductSubcategory
Tables               ProductVendor
Tables               SpecialOfferProduct
UserDefinedFunctions ufnGetProductDealerPrice
UserDefinedFunctions ufnGetProductListPrice
UserDefinedFunctions ufnGetProductStandardCost
Views                vProductAndDescription
Views                vProductModelCatalogDescription
Views                vProductModelInstructions
XmlSchemaCollections ProductDescriptionSchemaCollection
```

How it works...

After creating our usual SMO Server object, we create an SMO database handle to our `AdventureWorks2008R2` database.

```
$databasename = "AdventureWorks2008R2"
$db = $server.Databases[$databasename]
```

We also define our search string. Our goal is to get all database objects that have the word "Product" in their names:

```
#what keyword are we looking for?
$searchString = "Product"
```

We also create an empty array, where we can save our search results as records. This will enable us to display our final results in a tabular fashion when we're done with our iteration.

```
$results = @()
```

We will then go through all the database-related SMO properties and look for objects that contain the keyword we're looking for. Note that the following script will work only with PowerShell V3, because of the simplified `Where` cmdlet usage. If you want to use this in PowerShell V2, replace the `Where` syntax with the V2 variation.

```
#now we will loop through all database SMO
#properties and look of objects that match
#the search string
#note we are explicitly excluding Federations, because
#this throws an error
$db |
Get-Member -MemberType Property |
Where Definition -Like "*Smo*" |
Where Definition -NotLike "*Federation*" |
ForEach-Object {
   $type = $_.Name
   $db.$type |
   Where Name -Like "*$searchstring*" |
   ForEach-Object {
       $result = New-Object -Type PSObject -Prop @{
                   "ObjectType"=$type.Replace("Microsoft.SqlServer.
Management.Smo.", "")
                   "ObjectName"=$_.Name
                 }
       $results += $result
    }
}
```

In our loop, we have one long line that parses and creates our result.

The first part inspects each property and checks whether the name contains our search string.

```
$db |
Get-Member -MemberType Property |
Where Definition -Like "*Smo*" |
Where Definition -NotLike "*Federation*" |
ForEach-Object {
   $type = $_.Name
   $db.$type |
   Where Name -Like "*$searchstring*" |
   ForEach-Object {
       $result = New-Object -Type PSObject -Prop @{
                   "ObjectType"=$type.Replace("Microsoft.SqlServer.
Management.Smo.", "")
                   "ObjectName"=$_.Name
                 }
```

```
            $results += $result
        }
    }
```

Note that we have two conditions that we pass in the outer `Where-Object` cmdlets (here simplified to `Where` usage, which is supported only in PowerShell V3), as follows:

- `Where Definition -Like "*Smo*"`, because we are only looking for SMO properties
- `Where Definition -NotLike "*Federation*"`, because when you access `$db.Federations`, an exception is thrown

The second part builds a new row for the result with two columns: `ObjectType` and `ObjectName`. This new result is of type `PSObject`. Once constructed, we store this in our `$results` array. We also strip out the substring `Microsoft.SqlServer.Management.Smo` from the resulting object types, for brevity.

```
$db |
Get-Member -MemberType Property |
Where Definition -Like "*Smo*" |
Where Definition -NotLike "*Federation*" |
ForEach-Object {
   $type = $_.Name
   $db.$type |
   Where Name -Like "*$searchstring*" |
   ForEach-Object {
       $result = New-Object -Type PSObject -Prop @{
                    "ObjectType"=$type.Replace("Microsoft.SqlServer.
Management.Smo.", "")
                    "ObjectName"=$_.Name
                 }
       $results += $result
    }
}
```

Lastly, we export our results to a CSV file, using the `Export-Csv` cmdlet, and display in notepad:

```
#export results to csv file
$file = "C:\Temp\SearchResults.csv"
$results | Export-Csv -Path $file -NoTypeInformation

#display file contents
notepad $file
```

When you inspect your results, however, you will notice two extra objects that were not captured in our T-SQL statement in the *Getting ready* section. If we compare the two approaches, our PowerShell approach is more complete. In addition to the expected 23 results, PowerShell has also captured:

- `Production`—schema object
- `ProductDescriptionSchemaCollection`—`XmlSchemaCollection` object

There's more...

Another way to iterate through the objects is by using the `EnumObjects` method of the SMO database variable `$db`:

```
$searchString = "Product"

$db.EnumObjects() |
Where Name -Like "*$searchString*" |
Select DatabaseObjectTypes, Name |
Format-Table -AutoSize
```

Yes, there is still yet another alternative. This one is longer and less flexible, but it still gets you what you need. You can look for objects that match the search string by going through the `$db` object properties one by one, like this:

```
#long version is to enumerate explicitly each object type
$db.Tables | Where Name -Like "*$searchstring*"
$db.StoredProcedures  | Where Name -Like "*$searchstring*"
$db.Triggers  | Where Name -Like "*$searchstring*"
$db.UserDefinedFunctions  | Where Name -Like "*$searchstring*"

#etc
```

This is useful, and will be faster, if you know exactly what type of object you are looking for.

See also

- The *Exploring SMO Server objects* recipe in *Chapter 1*

Creating a database

This recipe walks through creating a database with default properties using PowerShell.

Getting ready

In this example, we are going to create a database called `TestDB`, and we assume that this database does not yet exist in your instance.

For your reference, the equivalent T-SQL code for this task is:

```
CREATE DATABASE TestDB
```

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
#database TestDB with default settings
#assumption is that this database does not yet exist
$dbName = "TestDB"
$db = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Database($server, $dbName)
$db.Create()

#to confirm, list databases in your instance
$server.Databases |
Select Name, Status, Owner, CreateDate
```

How it works...

There are two key steps to creating a database using SMO and PowerShell: creating an SMO `Server` object and creating an SMO `Database` object.

```
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName

$dbName = "TestDB"
$db = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Database($server, $dbName)
```

The SMO `Database` constructor requires both the SMO Server handle and a database object. The final action is to call the database object's `Create` method:

```
$db.Create()
```

Many SMO objects are consistent with the methods. You will see the `Create` method again in several recipes in this chapter.

Altering database properties

This recipe shows you how to change database properties, using SMO and PowerShell.

Getting ready

Create a database called `TestDB` by following the steps in the *Creating a database* recipe.

Using `TestDB`, we will:

- Change **ANSI NULLS Enabled** to **False**
- Change **ANSI PADDING Enabled** to **False**
- Restrict user access to **RESTRICTED_USER**
- Set the database to **Read Only**

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking
```

```
#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run

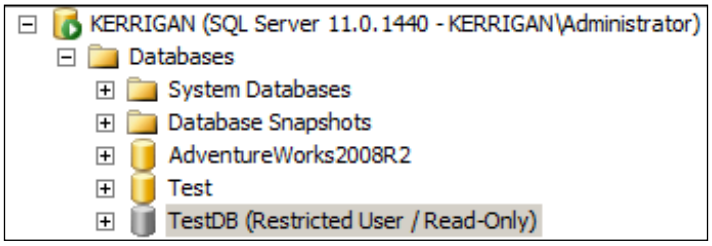
```
#database
$dbName = "TestDB"

#we are going to assume db exists
$db = $server.Databases[$dbName]

#DatabaseOptions
#change ANSI NULLS and ANSI PADDING
$db.DatabaseOptions.AnsiNullsEnabled = $false
$db.DatabaseOptions.AnsiPaddingEnabled = $false

#Change database access
#DatabaseUserAccess enum values: multiple, restricted, single
$db.DatabaseOptions.UserAccess = [Microsoft.SqlServer.Management.
Smo.DatabaseUserAccess]::Restricted

$db.Alter()


#some options are not available through the
#DatabaseOptions property
#so we will need to access the database object directly

#change compatiblity level to SQL Server 2005
#available CompatibilityLevel values are from
#Version 6.5 ('Version65') all the way to SQL
#Server 2012 ('Version110')
#however Version80 is not a valid compatibility option
#for SQL Server 2012
$db.AutoUpdateStatisticsEnabled = $true
$db.CompatibilityLevel = [Microsoft.SqlServer.Management.Smo.
CompatibilityLevel]::Version90
$db.Alter()

#set to readonly
$db.DatabaseOptions.ReadOnly = $true
$db.Alter()
```

4. Confirm the changes.

   To start confirming:

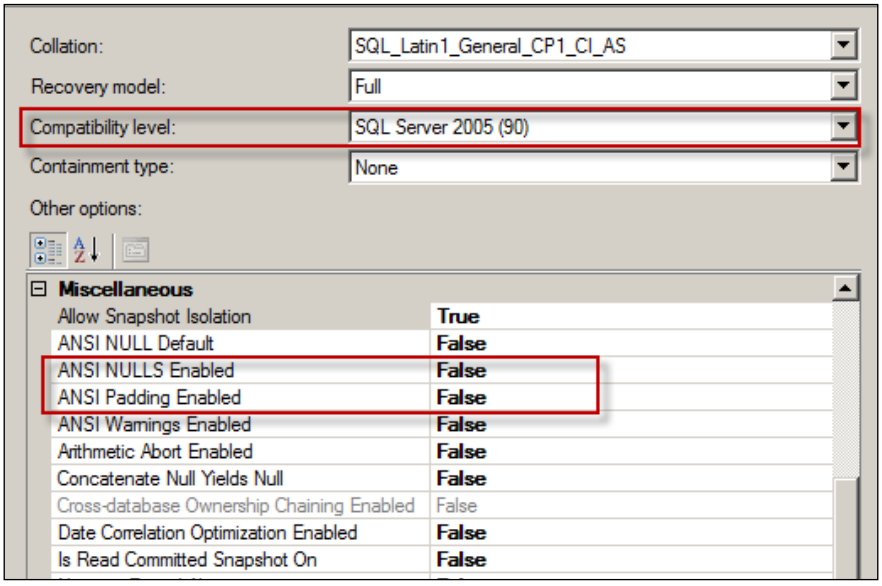
   1. Go to **Management Studio**.
   2. Connect to your instance.

   You will notice right away in **Object Explorer** that your database is grayed out and that its status has changed to **(Restricted User / Read-Only)**.



   To confirm **ANSI NULLS**, **ANSI PADDING**, and **Compatibility Level**:

   3. Right-click on the **TestDB** database and select **Properties**.
   4. Go to the **Options** tab, and check whether the respective options have been changed:



How it works...

To alter database properties, you will need to create an SMO handle to your database:

```
#we are going to assume db exists
$db = $server.Databases[$dbName]
```

After this, you will need to investigate which of the properties contains the setting you want to change. For example, **ANSI NULLS**, **ANSI WARNINGS**, database access restriction options, and **Read Only** are available through the `DatabaseOptions` property of your database object:

```
#DatabaseOptions
#change ANSI NULLS and ANSI PADDING
$db.DatabaseOptions.AnsiNullsEnabled = $false
$db.DatabaseOptions.AnsiPaddingEnabled = $false

#Change database access
#DatabaseUserAccess enum values: multiple, restricted, single
$db.DatabaseOptions.UserAccess = [Microsoft.SqlServer.Management.Smo.
DatabaseUserAccess]::Restricted

#set to readonly
$db.DatabaseOptions.ReadOnly = $true
```

`AutoUpdateStatisticsEnabled` and `CompatibilityLevel` are their own properties, directly accessible from the `$db` object:

```
$db.AutoUpdateStatisticsEnabled = $true
$db.CompatibilityLevel = [Microsoft.SqlServer.Management.Smo.
CompatibilityLevel]::Version90
```

Note that for SQL Server 2012, the earliest version you can set the compatibility level to is SQL Server 2005 (Version 90).

Once you've set the new values, you can persist the changes by invoking the `Alter` method of your database object:

```
$db.Alter()
```

Finding exactly which property the settings you are looking for reside in is half the battle, so it's a great idea to familiarize yourself with the properties of the object you are changing. Technet and MSDN are great resources, as are books and numerous articles and blog posts. However, remember there is help at your fingertips. Remember that the `Get-Member` cmdlet is your friend. You can invoke the `Get-Member` cmdlet as follows:

```
$db | Get-Member
```

See also

- The *Changing SQL Server instance configurations* recipe

Dropping a database

This recipe shows how you can drop a database, using PowerShell and SMO.

Getting ready

This task assumes you have created a database called `TestDB`. If you haven't, create one by following the steps in the *Creating a database* recipe.

How to do it...

The following are the steps to drop your `TestDB` database:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

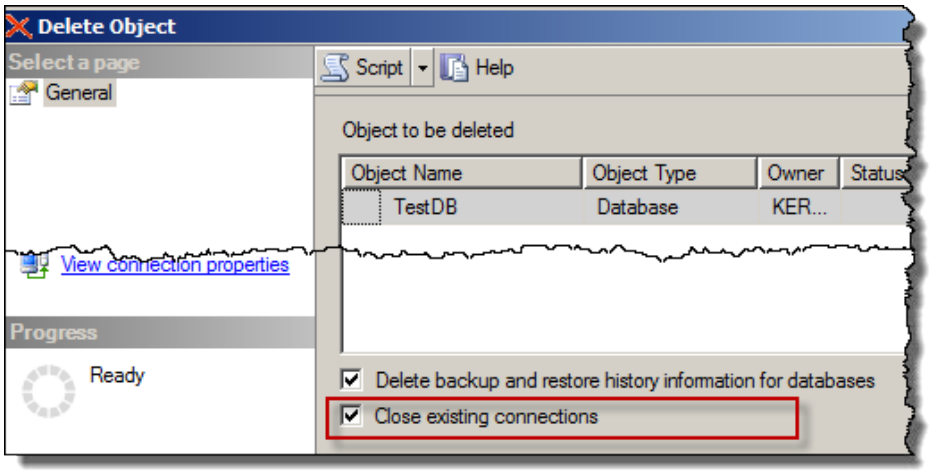
```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
$dbName = "TestDB"

#need to check if database exists, and if it does, drop it
$db = $server.Databases[$dbName]
if ($db)
{
      #we will use KillDatabase instead of Drop
      #Kill database will drop active connections before
      #dropping the database
      $server.KillDatabase($dbName)
}
```

How it works...

To drop an SMO server or database object, you can simply invoke the `Drop` method. However, if you have ever tried dropping a database before, you might have already experienced being blocked by active connections to that database. For this reason, we chose the `KillDatabase` method, which will kill active connections before dropping the database. This option is also available in Management Studio when you drop a database from **Object Explorer**. When you right-click on a database, the **Delete Object** window will appear. At the bottom of the window you will find a checkbox called **Close existing connections**, which will do the job.



Changing a database owner

This recipe shows how to programmatically change a SQL Server database owner.

Getting ready

This task assumes you have created a database called `TestDB` and that a Windows account `QUERYWORKS\aterra`. `QUERYWORKS\aterra` has been created in your test VM.

See *Appendix D, Creating a SQL Server VM*.

If you don't already have one, create a `TestDB` database by following the steps the *Creating a database* recipe.

How to do it...

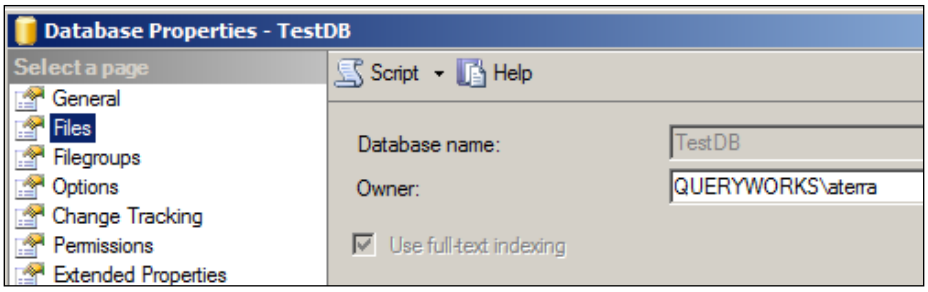
1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
#create database handle
$dbName = "TestDB"
$db = $server.Databases[$dbName]

#display current owner
$db.Owner

#change owner
#SetOwner requires two parameters:
#loginName and overrideIfAlreadyUser
$db.SetOwner("QUERYWORKS\aterra", $true)
#refresh db
$db.Refresh()

#check Owner value
$db.Owner
```

4. Do a visual check:
    1. Open **Management Studio**.
    2. Locate the **AdventureWorks2008R2** database.
    3. Right-click and go to **Properties**.
    4. Select **Options**.



How it works...

Changing the database owner is a short and straightforward task in PowerShell. First, you need to create a database handle.

The only other action required is invoking the `SetOwner` method of the `Microsoft.SqlServer.Management.Smo.Database` class, which requires two parameters:

- `LoginName`
- `OverrideIfAlreadyUser`

The `OverrideIfAlreadyUser` option can be set to either `true` or `false`. If set to `true`, it means that the currently logged-in user already exists as a user in the target database, and that user is dropped and re-added as owner. If set to `false` and the logged-in user is already mapped to that database, the `SetOwner` method will produce an error.

See also

- The *Altering database properties* recipe

Creating a table

This recipe shows how to create a table using PowerShell and SMO.

Getting ready

We will use the `AdventureWorks2008R2` database to create a table named `Student`, which has five columns. To give you a better idea of what we are trying to achieve, the equivalent T-SQL script needed to create this table is as follows:

```
USE AdventureWorks2008R2
GO
CREATE TABLE [dbo].[Student](
```

```
[StudentID] [INT] IDENTITY(1,1) NOT NULL,
[FName] [VARCHAR](50) NULL,
[LName] [VARCHAR](50) NOT NULL,
[DateOfBirth] [DATETIME] NULL,
[Age]  AS (DATEPART(YEAR,GETDATE())-DATEPART(YEAR,[DateOfBirth])),
CONSTRAINT [PK_Student_StudentID] PRIMARY KEY CLUSTERED
(
 [StudentID] ASC
)

GO
```

How to do it...

Let's create the `Student` table using PowerShell:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Next, add code to set up the database and table names and to drop the table if it already does exist:

```
$dbName = "AdventureWorks2008R2"
$tableName = "Student"
$db = $server.Databases[$dbName]
$table = $db.Tables[$tableName]

#if table exists drop
if($table)
{
   $table.Drop()
}
```

4. Add the following script to create the table, and run it:

```
#table class on MSDN
#http://msdn.microsoft.com/en-us/library/ms220470.aspx
```

```
$table = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Table -ArgumentList $db, $tableName

#column class on MSDN
#http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.
management.smo.column.aspx
#column 1
$col1Name = "StudentID"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::Int;
$col1 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList $table, $col1Name, $type
$col1.Nullable = $false
$col1.Identity = $true
$col1.IdentitySeed = 1
$col1.IdentityIncrement = 1
$table.Columns.Add($col1)

#column 2 - nullable
$col2Name = "FName"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::VarChar(50)
$col2 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList  $table, $col2Name, $type
$col2.Nullable = $true
$table.Columns.Add($col2)

#column 3 - not nullable, with default value
$col3Name = "LName"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::VarChar(50)
$col3 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList  $table, $col3Name, $type
$col3.Nullable = $false
$col3.AddDefaultConstraint("DF_Student_LName").Text = "'Doe'"

$table.Columns.Add($col3)

#column 4 - nullable, with default value
$col4Name = "DateOfBirth"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::DateTime;
$col4 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList  $table, $col4Name, $type
$col4.Nullable = $true
$col4.AddDefaultConstraint("DF_Student_DateOfBirth").Text =
"'1800-00-00'"
$table.Columns.Add($col4)
```

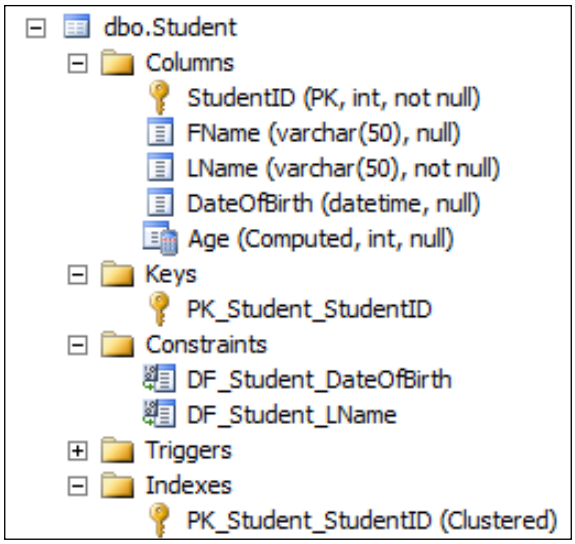
```
#column 5
$col5Name = "Age"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::Int;
$col5 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList  $table, $col5Name, $type
$col5.Nullable = $false
$col5.Computed = $true
$col5.ComputedText = "YEAR(GETDATE()) - YEAR(DateOfBirth)";
$table.Columns.Add($col5)

$table.Create()
```

5. Make `StudentID` the primary key, as follows:

```
########################################
#make StudentID a clustered PK
########################################
#note this is just a "placeholder" right now for PK
#no columns are added in this step
$PK=New-Object-TypeNameMicrosoft.SqlServer.Management.SMO.Index
-ArgumentList$table,"PK_Student_StudentID"
$PK.IsClustered =$true
$PK.IndexKeyType =[Microsoft.SqlServer.Management.SMO.
IndexKeyType]::DriPrimaryKey

#identify columns part of the PK
$PKcol=New-Object-TypeNameMicrosoft.SqlServer.Management.SMO.
IndexedColumn-ArgumentList$PK,$col1Name
$PK.IndexedColumns.Add($PKcol)
$PK.Create()
```

6. Do a visual check to see whether the table has been created with the correct columns and constraints:

    1. Open **Management Studio**.
    2. Go to the **AdventureWorks2008R2** database and expand **Tables**.
    3. Expand **Columns**, **Keys**, **Constraints**, and **Indexes**.



How it works...

To create a table, the first step is to create an SMO table object, thus:

```
$table = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Table -ArgumentList $db, $tableName
```

After this, all columns have to be defined one by one and added to the table before the `Create` method of the `Microsoft.SqlServer.Management.SMO.Table` class is invoked.

Let's take this step by step. To create a column, we first need to identify the data type we are storing in the column and the properties of that column.

Column data types in SMO are defined in `Microsoft.SqlServer.Management.SMO.DataType`. Every T-SQL data type is pretty much represented in this enumeration. To use a data type, the format should be as follows:

```
[Microsoft.SqlServer.Management.SMO.DataType]::DataType
```

To create a column, you will have to specify the table variable, the data type, and the column name:

```
$col1Name = "StudentID"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::Int
$col1 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList $table, $col1Name, $type
```

Common column properties will now be accessible to your column variable. Some common properties include:

- `Nullable`
- `Computed`
- `ComputedText`
- `Default Constraint` (by using the `AddDefaultConstraint` method)

For example:

```
#column 4 - nullable, with default value
$col4Name = "DateOfBirth"
$type = [Microsoft.SqlServer.Management.SMO.DataType]::DateTime;
$col4 =  New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Column -ArgumentList  $table, $col4Name, $type
$col4.Nullable = $true
$col4.AddDefaultConstraint("DF_Student_DateOfBirth").Text = "'1800-00-
00'"
```

There are additional properties that are exposed, depending on the data type you've chosen. For example, `[Microsoft.SqlServer.Management.SMO.DataType]::Int` will allow you to specify whether this is an identity and let you set seed and increment. `[Microsoft.SqlServer.Management.SMO.DataType]::Varchar` will allow you to set length.

Once you have set the properties, you can add columns to your table, as follows:

```
$table.Columns.Add($col4)
```

When everything is set up, you can invoke the table's `Create` method:

```
$table.Create()
```

Now, to create a primary key, you will need to create two other SMO Objects. The first one is the `Index` object. For this object, you can specify what type of index this is and whether it is clustered or nonclustered:

```
$PK = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Index -ArgumentList $table, "PK_Student_StudentID"
$PK.IsClustered = $true
$PK.IndexKeyType = [Microsoft.SqlServer.Management.SMO.
IndexKeyType]::DriPrimaryKey
```

The second object, `IndexedColumn`, specifies what columns are part of the index.

```
#identify columns part of the PK
$PKcol = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
IndexedColumn -ArgumentList $PK, $col1Name
```

If this column is an included column, simply set the `IsIncluded` property of the `IndexedColumn` object to `true`.

Once you've created all index columns, you can add them to the `Index` and invoke the `Create` method of the `Index` object:

```
$PK.IndexedColumns.Add($PKcol)
$PK.Create()
```

You must be thinking right now that what we've just gone over is a long-winded way to create a table. And you're thinking right. It is a more verbose way to create a table. However, keep in mind this is just *one more* way to get things done. When you need to create a table and if T-SQL is a faster way to do it, go for it. However, knowing how to do it in PowerShell and SMO is just one more tool in your arsenal for those scenarios where you might need to create the tables dynamically or more flexibly—for example, if you need to import the definition stored in Excel, CSV, or XML files from multiple users.

See also

- The *Creating an index* recipe
- Check out the complete list of SMO `DataType` classes from MSDN:

  `http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.datatype.aspx`

Creating a view

This recipe shows how to create a view using PowerShell and SMO.

Getting ready

We will use the `Person.Person` table in the `AdventureWorks2008R2` database for this recipe.

To give you an idea of what we are attempting to create in this recipe, this is the T-SQL equivalent:

```
CREATE VIEW dbo.vwVCPerson
AS
SELECT
   TOP 100
 BusinessEntityID,
 LastName,
 FirstName
```

```
FROM
 Person.Person
WHERE
    PersonType = 'IN'
ORDER BY
    LastName
GO
```

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

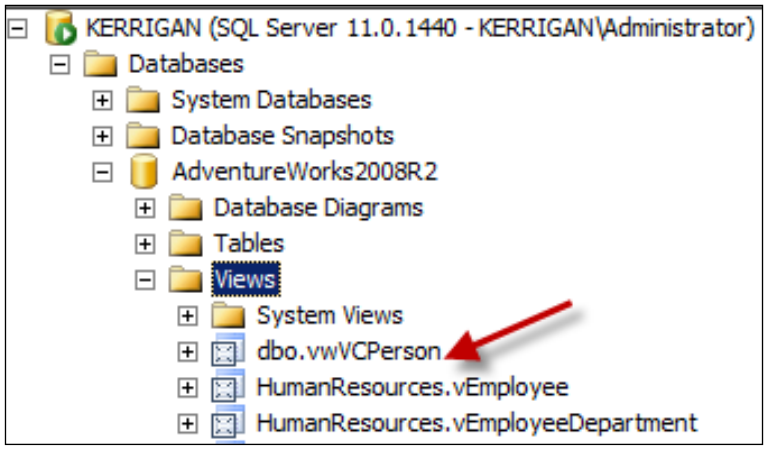
```
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]
$viewName = "vwVCPerson"
$view = $db.Views[$viewName]

#if view exists, drop it
if ($view)
{
   $view.Drop()
}

$view = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
View -ArgumentList $db, $viewName, "dbo"

#TextMode = false meaning we are not
#going to explicitly write the CREATE VIEW header
$view.TextMode = $false
$view.TextBody = @"
```

```
SELECT
    TOP 100
 BusinessEntityID,
 LastName,
 FirstName
FROM
 Person.Person
WHERE
    PersonType = 'IN'
ORDER BY
    LastName
"@

$view.Create()
```

4. Test the view from PowerShell by running the following code:

```
$result = Invoke-Sqlcmd `
-Query "SELECT * FROM vwVCPerson" `
-ServerInstance "$instanceName" `
-Database $dbName

$result | Format-Table -AutoSize
```

5. Do a visual check to see whether the view has been created. Open **Management Studio**, go to the **AdventureWorks2008R2** database, and expand **Views**.



How it works...

To create a view using SMO and PowerShell, you first need to create an SMO `View` variable, which requires three parameters: `database handle`, `view name`, and `schema`.

```
$view = New-Object -TypeName Microsoft.SqlServer.Management.SMO.View
-ArgumentList $db, $viewName, "dbo"
```

You can optionally set the view owner:

```
$view.Owner = "QUERYWORKS\aterra"
```

The crux of the view creation is with the view definition. You have the option here of setting the `TextMode` property to either `true` or `false`.

```
$view.TextMode = $false
$view.TextBody = @"
SELECT
    TOP 100
 BusinessEntityID,
 LastName,
 FirstName
FROM
 Person.Person
WHERE
    PersonType = 'IN'
ORDER BY
    LastName
"@
```

If you set the `TextMode` property to `false`, it means you are letting SMO construct the view header for you:

```
$view.TextMode = $false
```

If you set the `TextMode` property to `true`, it means you have to define the view's `TextHeader` property:

```
$view.TextMode = $true
$view.TextHeader = "CREATE VIEW dbo.vwVCPerson AS "
```

When all the pieces are in place, you can invoke the view's `Create` method:

```
$view.Create()
```

There's more...

When creating database objects such as views, stored procedures, or functions, you are often required to write blocks of code for the object definition. Although you can technically put all these in one line, it is best to put them in a multiline format for readability.

To embed these blocks of code in PowerShell, you will need to use a here-string. A here-string starts with `@"` followed by nothing else, and is ended by `"@`, which must be the first two character in its own line:

```
$view.TextBody = @"
SELECT
    TOP 100
 BusinessEntityID,
 LastName,
 FirstName
FROM
 Person.Person
WHERE
    PersonType = 'IN'
ORDER BY
    LastName
"@
```

This construction might remind you a little bit of a C-style comment, which starts with `/*` and ends with `*/`, albeit using different characters.

Creating a stored procedure

This recipe shows how to create an encrypted stored procedure using SMO and PowerShell.

Getting ready

The T-SQL equivalent of the encrypted stored procedure we are about to recreate in PowerShell is as follows:

```
CREATE PROCEDURE [dbo].[uspGetPersonByLastName] @LastName [varchar]
(50)
WITH ENCRYPTION
AS
```

```
SELECT
   TOP 10
   BusinessEntityID,
   LastName
FROM
   Person.Person
WHERE
   LastName = @LastName
```

How to do it...

Follow these steps to create the `uspGetPersonByLastName` stored procedure using PowerShell:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

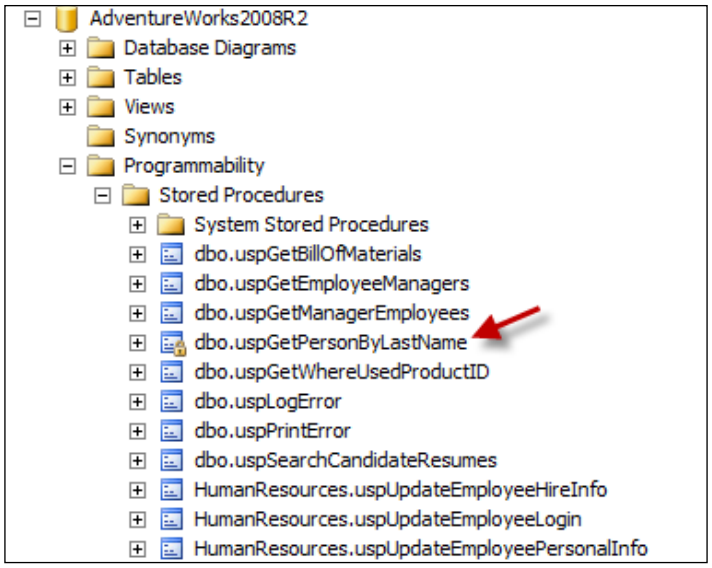
3. Add the following script and run it:

```
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]

#storedProcedure class on MSDN:
#http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.
management.smo.storedprocedure.aspx

$sprocName = "uspGetPersonByLastName"
$sproc = $db.StoredProcedures[$sprocName]
#if stored procedure exists, drop it
if ($sproc)
{
   $sproc.Drop()
}
```

```
$sproc = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
StoredProcedure -ArgumentList $db, $sprocName

#TextMode = false means stored procedure header
#is not editable as text
#otherwise our text will contain the CREATE PROC block
$sproc.TextMode = $false
$sproc.IsEncrypted = $true

$paramtype = [Microsoft.SqlServer.Management.SMO.
Datatype]::VarChar(50);
$param = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.StoredProcedureParameter -ArgumentList $sproc,"@
LastName",$paramtype
$sproc.Parameters.Add($param)

#Set the TextBody property to define the stored procedure.
$sproc.TextBody =  @"
SELECT
   TOP 10
   BusinessEntityID,
   LastName
FROM
   Person.Person
WHERE
   LastName = @LastName
"@

# Create the stored procedure on the instance of SQL Server.
$sproc.Create()

#if later on you need to change properties, can use the Alter
method
```

4. Do a visual check to see whether the stored procedure has been created.
    1. Open **Management Studio**.
    2. Go to the **AdventureWorks2008R2** database.

3. Expand **Programmability** | **Stored Procedures**.
4. Check that the stored procedure is there.



5. Test the stored procedure from PowerShell. In the same session, type the following code and run it:

```
$lastName = "Abercrombie"
$result = Invoke-Sqlcmd `
-Query "EXEC uspGetPersonByLastName @LastName=`'$LastName`'" `
-ServerInstance "$instanceName" `
-Database $dbName

$result | Format-Table -AutoSize
```

How it works...

To create a stored procedure, you first need to initialize an SMO `StoredProcedure` object. When creating this object, you need to pass the database handle and the stored procedure name as parameters:

```
$sproc = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
StoredProcedure -ArgumentList $db, $sprocName
```

You can then set some properties of the stored procedure object, such as whether it's encrypted or not:

```
$sproc.IsEncrypted = $true
```

If you specify `TextMode = true`, you will need to create the stored procedure header yourself. If you have parameters, these will have to be defined in your text header, for example:

```
$sproc.TextMode = $true
$sproc.TextHeader = @"
CREATE PROCEDURE [dbo].[uspGetPersonByLastName]
 @LastName [varchar](50)
AS
"@
```

Otherwise, if you set `TextMode = $false`, you are technically allowing PowerShell to autogenerate this header for you, based on the other properties and parameters you have set. You will also have to create the parameter objects one-by-one and add them to the stored procedure.

```
$sproc.TextMode = $false

$paramtype = [Microsoft.SqlServer.Management.SMO.
Datatype]::VarChar(50);
$param = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
StoredProcedureParameter -ArgumentList $sproc,"@LastName",$paramtype
$sproc.Parameters.Add($param)
```

When creating the stored procedure, use a here-string as you set the definition of the `TextBody` property of the stored procedure object:

```
$sproc.TextBody =  @"
SELECT
   TOP 10
   BusinessEntityID,
   LastName
FROM
   Person.Person
WHERE
   LastName = @LastName
"@
```

Once the header, definition, and properties of the stored procedure are in place, you can invoke the `Create` method, which sends the `CREATEPROC` statement to SQL Server and creates the stored procedure.

```
# Create the stored procedure on the instance of SQL Server.
$sproc.Create()
```

Creating a trigger

This recipe demonstrates how to programmatically create a trigger in SQL Server using SMO and PowerShell.

Getting ready

For this recipe, we will use the `Person.Person` table in the `AdventureWorks2008R2` database. We will create a trivial `AFTER` trigger that merely displays values from the inserted and deleted records upon firing.

The following is the T-SQL equivalent of what we are going to accomplish programmatically in this section:

```
CREATE TRIGGER [Person].[tr_u_Person]
ON [Person].[Person]
AFTER  UPDATE
AS

  SELECT
     GETDATE() AS UpdatedOn,
     SYSTEM_USER AS UpdatedBy,
     i.LastName AS NewLastName,
     i.FirstName AS NewFirstName,
     d.LastName AS OldLastName,
     d.FirstName AS OldFirstName
  FROM
     inserted i
     INNER JOIN deleted d
     ON i.BusinessEntityID = d.BusinessEntityID
```

How to do it...

Let's follow these steps to create an `AFTER` trigger in PowerShell:

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
```

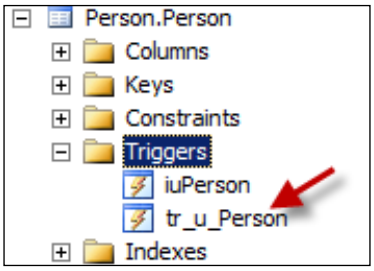
```
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]
$tableName = "Person"
$schemaName = "Person"

#get a handle to the Person.Person table
$table = $db.Tables |
         Where Schema -Like "$schemaName" |
         Where Name -Like "$tableName"

$triggerName = "tr_u_Person";
#note here we need to check triggers attached to table
$trigger = $table.Triggers[$triggerName]

#if trigger exists, drop it
if ($trigger)
{
   $trigger.Drop()
}

$trigger = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.Trigger -ArgumentList $table, $triggerName
$trigger.TextMode = $false

#this is just an update trigger
$trigger.Insert = $false
$trigger.Update = $true
$trigger.Delete = $false

#3 options for ActivationOrder: First, Last, None
$trigger.InsertOrder = [Microsoft.SqlServer.Management.SMO.Agent.
ActivationOrder]::None
$trigger.ImplementationType = [Microsoft.SqlServer.Management.SMO.
ImplementationType]::TransactSql

#simple example
$trigger.TextBody = @"
  SELECT
     GETDATE() AS UpdatedOn,
     SYSTEM_USER AS UpdatedBy,
```

```
 i.LastName AS NewLastName,
     i.FirstName AS NewFirstName,
     d.LastName AS OldLastName,
     d.FirstName AS OldFirstName
  FROM
     inserted i
     INNER JOIN deleted d
     ON i.BusinessEntityID = d.BusinessEntityID

"@

$trigger.Create()
```

4. Do a visual check to see whether the stored procedure has been created. Open **Management Studio**.



5. Test the stored procedure using PowerShell:

```
$firstName = "Frankk"
$result = Invoke-Sqlcmd `
-Query "UPDATE Person.Person  SET FirstName = `'$firstName`' WHERE
BusinessEntityID = 2081 " `
-ServerInstance "$instanceName" `
-Database $dbName

$result | Format-Table -AutoSize
```

Your result should look similar to the following:

```
UpdatedOn             UpdatedBy              NewLastName NewFirstName OldLastName OldFirstName
---------             ---------              ----------- ------------ ----------- ------------
12/25/2011 1:40:22 PM KERRIGAN\Administrator Zhang       Frankk       Zhang       Frank
```

How it works...

The code for this section is quite long, so we will break it down here.

To create a trigger, you need to create a reference to both the instance and the database first. This is something we have done for most of the recipes in this chapter, in case you have skipped the previous recipes.

A trigger is bound to a table or view. You will need to create a variable that points to the table you want the trigger to attach to:

```
$tableName = "Person"
$schemaName = "Person"

$table = $db.Tables |
         Where Schema -Like "$schemaName" |
         Where Name -Like "$tableName"
```

For purposes of this recipe, if the trigger exists, we will drop it.

```
$trigger = $table.Triggers[$triggerName]

#if trigger exists, drop it
if ($trigger)
{
   $trigger.Drop()
}
```

Next, you need to create an SMO Trigger object:

```
$trigger = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Trigger -ArgumentList $table, $triggerName
```

Next, set the `TextMode` property. If set to `true`, it means you have to define the trigger header text yourself. Otherwise, SMO will automatically generate it for you.

```
$trigger.TextMode = $false
```

You will also need to define what type of DML trigger this is. Your options are `insert`, `update`, and/or `delete` triggers. Our example is just an `update` trigger.

```
#this is just an update trigger
$trigger.Insert = $false
$trigger.Update = $true
$trigger.Delete = $false
```

You can also optionally define the trigger order. By default, there is no guarantee in what order the triggers will be run by SQL Server, but you have the option to set it to `First` or `Last`. In our example, we leave it at the default value, but we still explicitly define it for readability.

```
#3 options for ActivationOrder: First, Last, None
$trigger.InsertOrder = [Microsoft.SqlServer.Management.SMO.Agent.
ActivationOrder]::None
```

Our trigger is a Transact-SQL trigger. SQL Server SMO also supports SQLCLR triggers.

```
$trigger.ImplementationType = [Microsoft.SqlServer.Management.SMO.
ImplementationType]::TransactSql
```

To specify the trigger definition, you can set the value of the trigger's `TextBody` property. You can use a here-string to assign the trigger code block to the `TextBody` property:

```
#simple example
$trigger.TextBody = @"
  SELECT
     GETDATE() AS UpdatedOn,
     SYSTEM_USER AS UpdatedBy,
     i.LastName AS NewLastName,
     i.FirstName AS NewFirstName,
     d.LastName AS OldLastName,
     d.FirstName AS OldFirstName
  FROM
     inserted i
     INNER JOIN deleted d
     ON i.BusinessEntityID = d.BusinessEntityID

"@
```

When ready, invoke the `Create()` method of the trigger.

```
$trigger.Create()
```

Creating an index

This recipe demonstrates how to create a non-clustered index with an included column using PowerShell and SMO.

Getting ready

We will use the `Person.Person` table in the `AdventureWorks2008R2` database. We will create a non-clustered index on `FirstName`, `LastName`, and include `MiddleName`. The T-SQL equivalent of this task is:

```
CREATE NONCLUSTERED INDEX [idxLastNameFirstName]
ON [Person].[Person]
(
 [LastName] ASC,
 [FirstName] ASC
)
INCLUDE ( [MiddleName])
GO
```

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

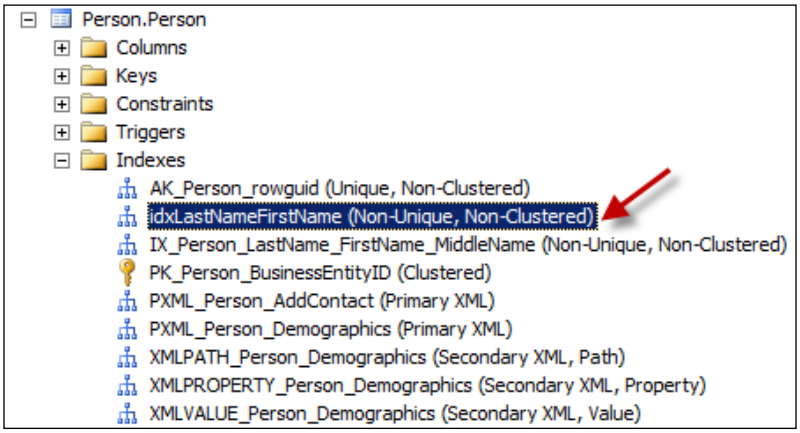
```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]

$tableName = "Person"
$schemaName = "Person"

$table = $db.Tables |
         Where Schema -Like "$schemaName" |
```

```
Where Name -Like "$tableName"


$indexName = "idxLastNameFirstName"
$index = $table.Indexes[$indexName]
#if stored procedure exists, drop it
if ($index)
{
   $index.Drop()
}

$index = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Index -ArgumentList $table, $indexName

#first index column, by default sorted ascending
$idxCol1 = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.IndexedColumn -ArgumentList $index, "LastName", $false
$index.IndexedColumns.Add($idxCol1)

#second index column, by default sorted ascending
$idxCol2 = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.IndexedColumn -ArgumentList $index, "FirstName", $false
$index.IndexedColumns.Add($idxCol2)

#included column
$inclCol1 = New-Object -TypeName Microsoft.SqlServer.Management.
SMO.IndexedColumn -ArgumentList $index, "MiddleName"
$inclCol1.IsIncluded = $true
$index.IndexedColumns.Add($inclCol1)

#Set the index properties.
<#
None            - no constraint
DriPrimaryKey - primary key
DriUniqueKey - unique constraint
#>
$index.IndexKeyType = [Microsoft.SqlServer.Management.SMO.
IndexKeyType]::None
$index.IsClustered = $false
$index.FillFactor = 70


#Create the index on the instance of SQL Server.
$index.Create()
```

4. Do a visual check to see whether the stored procedure has been created. Open **Management Studio:**



How it works...

The first step to creating an index is to create an SMO index object, which requires both the table/view handle and the index name:

```
$index = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
Index -ArgumentList $table, $indexName
```

The next step is to identify all index columns using the `IndexedColumn` property of the `Microsoft.SqlServer.Management.SMO.Index` class:

```
#first index column
$idxCol1 = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
IndexedColumn -ArgumentList $index, "LastName", $false; #sort asc
$index.IndexedColumns.Add($idxCol1)

#second index column
$idxCol2 = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
IndexedColumn -ArgumentList $index, "FirstName", $false; #sort asc
$index.IndexedColumns.Add($idxCol2)
```

Optionally, you can add included columns, in other words, columns that "tag along" with the index but are not part of the indexed columns:

```
#included column
$inclCol1 = New-Object -TypeName Microsoft.SqlServer.Management.SMO.
IndexedColumn -ArgumentList $index, "MiddleName"
$inclCol1.IsIncluded = $true
$index.IndexedColumns.Add($inclCol1)
```

The type of the index can be specified using the `IndexKeyType` property of the `Microsoft.SqlServer.Management.SMO.IndexedColumn` class, which accepts three possible values:

- `None`: Non-unique
- `DriPrimaryKey`: Primary key
- `DriUniqueKey`: Unique key

Additional properties can also be set, including `FillFactor`, and whether this key is clustered or not:

```
$index.IndexKeyType = [Microsoft.SqlServer.Management.SMO.
IndexKeyType]::None
$index.IsClustered = $false
$index.FillFactor = 70
```

When all properties are set, invoke the `Create` method of the SMO index object.

```
#Create the index on the instance of SQL Server.
$index.Create()
```

There's more...

The SMO Index object also supports different kinds of indexes:

Index Type	What to set
Filtered	`HasFilter`
	`FilterDefinition`
FullText	`IsFullTextKey` = `$true`
XML	`IsXMLIndex` = `$true`
Spatial	`IsSpatialIndex` = `$true`

To get more information about index options, check out the MSDN documentation on SMO indexes:

`http://msdn.microsoft.com/en-us/library/microsoft.sqlserver.management.smo.index.aspx`

See also

- The *Creating a table* recipe

Executing a query / SQL script

This recipe shows how you can execute either a hardcoded query or a SQL script, from PowerShell.

Getting ready

Create a file in your `C:\Temp` folder called `SampleScript.sql`. This should contain:

```
SELECT *
FROM Person.Person
```

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]

#execute a passthrough query, and export to a CSV file
Invoke-Sqlcmd `
-Query "SELECT * FROM Person.Person" `
-ServerInstance "$instanceName" `
-Database $dbName |
Export-Csv -LiteralPath "C:\Temp\ResultsFromPassThrough.csv" `
-NoTypeInformation

#execute the SampleScript.sql, and display results to screen
Invoke-SqlCmd `
-InputFile "C:\Temp\SampleScript.sql" `
-ServerInstance "$instanceName" `
-Database $dbName |
Select FirstName, LastName, ModifiedDate |
Format-Table
```

How it works...

Start warming up to the `Invoke-Sqlcmd` cmdlet. We will be using it a lot in this book.

As the name suggests, this cmdlet allows you to run T-SQL code or scripts and commands supported by the `SQLCMD` utility. It also allows you to run XQuery code. `Invoke-Sqlcmd` is your all-purpose SQL utility cmdlet.

To get more information about `Invoke-Sqlcmd`, use the `Get-Help` cmdlet

```
Get-Help Invoke-Sqlcmd -Full
```

In this recipe, we looked at two ways of using `Invoke-Sqlcmd`. The first is by specifying a query to run. For this, you should use the `–Query` option:

```
#execute a passthrough query, and export to a CSV file
Invoke-Sqlcmd `
-Query "SELECT * FROM Person.Person" `
-ServerInstance "$instanceName" `
-Database $dbName |
Export-Csv -LiteralPath "C:\Temp\ResultsFromPassThrough.csv" `
-NoTypeInformation
```

For the second way, which requires running a SQL Script, you need to specify the `–InputFile` switch:

```
#execute the SampleScript.sql, and display results to screen
Invoke-SqlCmd `
-InputFile "C:\Temp\SampleScript.sql" `
-ServerInstance "$instanceName" `
-Database $dbName |
Select FirstName, LastName, ModifiedDate |
Format-Table
```

Performing bulk export using Invoke-Sqlcmd

This recipe demonstrates how to export contents of a table to a CSV file using PowerShell and the `Invoke-Sqlcmd` cmdlet.

Getting ready

Make sure you have access to the `AdventureWorks2008R2` database. We will use the `Person.Person` table.

Create a `C:\Temp` folder, if you don't already have one on your system.

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Import the `SQLPS` module, and create a new SMO Server object:

```
#import SQL Server module
Import-Module SQLPS -DisableNameChecking

#replace this with your instance name
$instanceName = "KERRIGAN"
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.
Server -ArgumentList $instanceName
```

3. Add the following script and run it:

```
#database handle
$dbName = "AdventureWorks2008R2"
$db = $server.Databases[$dbName]

#export file name
$exportfile = "C:\Temp\Person_Person.csv"

$query = @"
SELECT
   *
FROM
   Person.Person
"@
Invoke-Sqlcmd -Query $query -ServerInstance "$instanceName"
-Database $dbName |
Export-Csv -LiteralPath $exportfile -NoTypeInformation
```

How it works...

In this recipe, we export the results of a query to a CSV file. There are two core parts of the export approach in this recipe.

The first part is executing the query, and for this, we use the `Invoke-Sqlcmd` cmdlet. We specify the instance and database and send a query to SQL Server through this cmdlet:

```
Invoke-Sqlcmd -Query $query -ServerInstance "$instanceName" -Database
$dbName |
Export-Csv -LiteralPath $exportfile -NoTypeInformation
```

The second part is piping the results to the `Export-Csv` cmdlet and specifying the file in which the results are supposed to be stored. We also specify `–NoTypeInformation`, so the cmdlet will omit the `#TYPE` .NET information type as the first line in the file:

```
Invoke-Sqlcmd -Query $query -ServerInstance "$instanceName" -Database
$dbName |
Export-Csv -LiteralPath $exportfile -NoTypeInformation
```

See also

- The *Executing a query / SQL script* recipe

Performing bulk export using bcp

This recipe demonstrates how to export contents of a table to a CSV file using PowerShell and `bcp`.

Getting ready

Make sure you have access to the `AdventureWorks2008R2` database. We will export the `Person.Person` table to a timestamped text file delimited by a pipe (`|`).

Create a `C:\Temp\Exports` folder, if you don't already have it on your system.

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Add the following script and run the following code:

```
$server = "KERRIGAN"
$table = "AdventureWorks2008R2.Person.Person"
$curdate = Get-Date -Format "yyyy-MM-dd_hmmtt"

$foldername = "C:\Temp\Exports\"

#format file name
$formatfilename = "$($table)_$($curdate).fmt"

#export file name
$exportfilename = "$($table)_$($curdate).csv"

$destination_exportfilename = "$($foldername)$($exportfilename)"
```

```
$destination_formatfilename = "$($foldername)$($formatfilename)"

#command to generate format file
$cmdformatfile = "bcp $table format nul -T -c -t `"|`" -r `"\n`"
-f `"$($destination_formatfilename)`" -S$($server)"

#command to generate the export file
$cmdexport = "bcp $($table) out `"$($destination_exportfilename)`"
-S$($server) -T -f `"$destination_formatfilename`""

<#
$cmdformatfile gives you something like this:
bcp AdventureWorks2008R2.Person.Person format nul -T -c -t "|" -r
"\n" -f "C:\Temp\Exports\AdventureWorks2008R2.Person.Person_2011-
12-27_913PM.fmt" -S KERRIGAN

$cmdexport gives you something like this:
bcp AdventureWorks2008R2.Person.Person out "C:\Temp\Exports\
AdventureWorks2008R2.Person.Person_2011-12-27_913PM.csv" -S
KERRIGAN -T -c -f "C:\Temp\Exports\AdventureWorks2008R2.Per
son.Person_2011-12-27_913PM.fmt"
#>

#run the format file command
Invoke-Expression $cmdformatfile

#delay 1 sec, give server some time to generate the format file
#sleep helps us avoid race conditions
Start-Sleep -s 1

#run the export command
Invoke-Expression $cmdexport

#check the folder for generated file
explorer.exe $foldername
```

How it works...

Using SQL Server's `bcp` command is often the faster way to export records out of SQL Server. It is also often preferred, because `bcp` offers flexibility in the export format.

The default export format of `bcp` uses a tab (`\t`) as a field delimiter and a carriage return newline character (`\r\n`) as a row delimiter. If you want to change this, you will need to create and use a format file that specifies how you want the export to be formatted.

In our recipe, we first timestamp both the format file and then export file names.

```
$curdate = Get-Date -Format "yyyy-MM-dd_hmmtt"

$foldername = "C:\Temp\Exports\"

#format file name
$formatfilename = "$($table)_$($curdate).fmt"

#export file name
$exportfilename = "$($table)_$($curdate).csv"

$destination_exportfilename = "$($foldername)$($exportfilename)"
$destination_formatfilename = "$($foldername)$($formatfilename)"
```

We then construct the string that will generate the format file as follows:

```
#command to generate the export file
$cmdexport = "bcp $($table) out `"$($destination_exportfilename)`"
-S$($server) -T -f `"$destination_formatfilename`""
```

Note that because the actual command requires double quotes, when we construct the command, we need to escape the double quote within the command with a backtick (`).

This command that is constructed should be similar to the following:

```
bcp AdventureWorks2008R2.Person.Person format nul -T -c -t "|" -r
"\n" -f "C:\Temp\Exports\AdventureWorks2008R2.Person.Person_2011-12-
27_913PM.fmt" -SKERRIGAN
```

We also construct the command that will export the records using the format file we just created:

```
#command to generate the export file
$cmdexport = "bcp $($table) out `"$($destination_exportfilename)`"
-S$($server) -T -f `"$destination_formatfilename`""
```

This will give us something similar to the following:

```
bcp AdventureWorks2008R2.Person.Person out "C:\Temp\Exports\
AdventureWorks2008R2.Person.Person_2011-12-27_913PM.csv" -SKERRIGAN
-T -f "C:\Temp\Exports\AdventureWorks2008R2.Person.Person_2011-12-
27_913PM.fmt"
```

When the strings containing the commands are complete, we can execute the command using the `Invoke-Expression` cmdlet. We run the format file creation command first, and then use the `Start-Sleep` cmdlet to pause for 1 second, to ensure the format file has been created first, before we invoke the command to do the actual export.

```
#run the format file command
Invoke-Expression $cmdformatfile

#delay 1 sec, give server some time to generate
#the format file
#sleep helps us avoid race conditions
Start-Sleep -s 1

#run the export command
Invoke-Expression $cmdexport
```

If we don't wait, there will be a bigger chance for all the commands to be executed really fast, and the command to export will run before the format file has been generated. This will lead to an error, because the `bcp` command will not be able to find the format file.

Lastly, we just open up Windows Explorer, so we can inspect the files we generated.

```
#check the folder for generated file
explorer.exe $foldername
```

See also

- The *Performing bulk export using Invoke-Sqlcmd* recipe
- Read more about `bcp` format file options at `http://msdn.microsoft.com/en-us/library/ms191516.aspx`.

Performing bulk import using BULK INSERT

This recipe will walk you through importing contents of a CSV file to SQL Server using PowerShell and `BULK INSERT`.

Getting ready

To do a test import, we will first need to create a `Person` table similar to the `Person.Person` table from the `AdventureWorks2008R2` database, with some slight modifications.

We will create this in the `Test` schema, and we will remove some of the constraints and keep this table as simple and independent as we can.

To create the table that we need for this exercise, open up **Management Studio** and run the following code:

```
CREATE SCHEMA [Test]
GO
```

```
CREATE TABLE [Test].[Person](
 [BusinessEntityID] [int] NOT NULL PRIMARY KEY,
 [PersonType] [nchar](2) NOT NULL,
 [NameStyle] [dbo].[NameStyle] NOT NULL,
 [Title] [nvarchar](8) NULL,
 [FirstName] [dbo].[Name] NOT NULL,
 [MiddleName] [dbo].[Name] NULL,
 [LastName] [dbo].[Name] NOT NULL,
 [Suffix] [nvarchar](10) NULL,
 [EmailPromotion] [int] NOT NULL,
 [AdditionalContactInfo] [xml] NULL,
 [Demographics] [xml] NULL,
 [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,
 [ModifiedDate] [datetime] NOT NULL
)

GO
```

For this recipe, we will import a file called `AdventureWorks2008R2.Person.Person.csv`, which is provided with the downloadable materials from the Packt site. Save this in the folder `C:\Temp\Exports`.

Alternatively, create a CSV file, as mentioned in the *Performing bulk export using bcp* recipe, and replace the filename reference in this recipe with the filename you generate.

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Let's add some helper functions first. Type the following and execute it:

```
Import-Module SQLPS -DisableNameChecking

function Import-Person {
<#
.SYNOPSIS
    Very simple function to get number
    of records in Test.Person
.NOTES
    Author     : Donabel Santos
.LINK
    http://www.sqlmusings.com
#>
param([string]$instanceName,[string]$dbName)
```

```
$query = @"
TRUNCATE TABLE Test.Person
GO
BULK INSERT AdventureWorks2008R2.Test.Person
   FROM 'C:\Temp\Exports\AdventureWorks2008R2.Person.Person.csv'
   WITH
      (
         FIELDTERMINATOR ='|',
         ROWTERMINATOR ='\n'
      )
SELECT COUNT(*) AS NumRecords
FROM AdventureWorks2008R2.Test.Person
"@;

#check number of records
Invoke-Sqlcmd -Query $query `
-ServerInstance "$instanceName" `
-Database $dbName
}
```

3. Now let's invoke the function in the same session, as follows:

```
$instanceName = "KERRIGAN"
$dbName = "AdventureWorks2008R2"
Import-Person $instanceName $dbName
```

How it works...

Importing records from a CSV or text file into a SQL Server table using the `BULK INSERT` command will require constructing the `BULK INSERT` T-SQL statement and executing this statement using the `Invoke-Sqlcmd` cmdlet:

```
Invoke-Sqlcmd -Query $query `
-ServerInstance "$instanceName" `
-Database $dbName
```

However, we have done things a little bit differently than in our previous recipes. In this recipe, we first created a function that encapsulates all the core import tasks.

To create a function, you first need to create a function header:

```
function Import-Person {
```

The function header starts with the keyword `function` and is then followed by the function name in the format *verb-noun*. The body of the function is encapsulated by opening and closing curly braces `{ }`.

Right after the function header, we also create a comment-based help header comment.

```
<#
.SYNOPSIS
    Very simple function to get number      of records in Test.Person
.NOTES
    Author      : Donabel Santos
.LINK
    http://www.sqlmusings.com
#>
```

Block comments in PowerShell start with `<#` and end with `#>`. In addition, this is a special type of block comment that allows this function's comments to be displayed in a `Get-Help` cmdlet. We now type:

```
Get-Help Import-Person
```

This will provide output similar to the help you get for any other cmdlet:

```
PS C:\Users\Administrator> Get-Help Import-Person

NAME
    Import-Person

SYNOPSIS
    Very simple function to get number
    of records in Test.Person


SYNTAX
    Import-Person [[-instanceName] <String>] [[-dbName] <String>] [<CommonParameters>]


DESCRIPTION


RELATED LINKS
     http://www.sqlmusings.com

REMARKS
    To see the examples, type: "get-help Import-Person -examples".
    For more information, type: "get-help Import-Person -detailed".
    For technical information, type: "get-help Import-Person -full".
```

After the function header and comment come the parameters. Our `Import-Person` function accepts two parameters: `instance name` and `database name`.

```
param([string]$instanceName,[string]$dbName)
```

Following our parameter definition is the function definition. We start by creating a here-string, which contains our T-SQL statement:

```
$query = @"
TRUNCATE TABLE Test.Person
GO
BULK INSERT AdventureWorks2008R2.Test.Person
   FROM 'C:\Temp\Exports\AdventureWorks2008R2.Person.Person.csv'
   WITH
      (
         FIELDTERMINATOR ='|',
         ROWTERMINATOR ='\n'
      )
SELECT COUNT(*) AS NumRecords
FROM AdventureWorks2008R2.Test.Person
"@;
```

After our query is constructed, we pass it to the `Invoke-Sqlcmd` cmdlet, which in turn sends and executes it in our SQL Server instance.

```
Invoke-Sqlcmd -Query $query `
-ServerInstance "$instanceName" `
-Database $dbName
```

Functions in PowerShell are local-scoped by default, but when run through the ISE maintain a global scope. In our recipe, once you run the first part of the script that has the function definition, this function can be invoked at any time in the current session. We can see that the function simplifies importing the records and all that we need is the instance name, the database name, and the `Import-Person` function.

```
$instanceName = "KERRIGAN"
$dbName = "AdventureWorks2008R2"
Import-Person $instanceName $dbName
```

If you are using the shell and you want this function to persist globally across different scopes, save the script as a `.ps1` file and dot source it. Another way is to prepend the function name with `global`:

```
function global:Import-Person {
```

See also

- The *Executing a query / SQL script* recipe
- The *Performing bulk import using bcp* recipe

Performing bulk import using bcp

This recipe will walk you through the process of importing the contents of a CSV file to SQL Server using PowerShell and `bcp`.

Getting ready

To do a test import, let's first create a `Person` table similar to the `Person.Person` table from the `AdventureWorks2008R2` database, with some slight modifications. We will create this in the `Test` schema, and we will remove some of the constraints and keep this table as simple and independent as we can.

If `Test.Person` does not yet exist in your environment, let's create it. Open up **Management Studio**, and run the following code:

```
CREATE SCHEMA [Test]
GO
CREATE TABLE [Test].[Person](
 [BusinessEntityID] [int] NOT NULL PRIMARY KEY,
 [PersonType] [nchar](2) NOT NULL,
 [NameStyle] [dbo].[NameStyle] NOT NULL,
 [Title] [nvarchar](8) NULL,
 [FirstName] [dbo].[Name] NOT NULL,
 [MiddleName] [dbo].[Name] NULL,
 [LastName] [dbo].[Name] NOT NULL,
 [Suffix] [nvarchar](10) NULL,
 [EmailPromotion] [int] NOT NULL,
 [AdditionalContactInfo] [xml] NULL,
 [Demographics] [xml] NULL,
 [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,
 [ModifiedDate] [datetime] NOT NULL
)

GO
```

How to do it...

1. Open the PowerShell console by going to **Start** | **Accessories** | **Windows PowerShell** | **Windows PowerShell ISE**.
2. Let's add some helper functions first. Type the following and then run it:

```
Import-Module SQLPS -DisableNameChecking
$instanceName = "KERRIGAN"
$dbName = "AdventureWorks2008R2"
```

```
function Truncate-Table {
<#
.SYNOPSIS
    Very simple function to truncate
    records from Test.Person
.NOTES
    Author     : Donabel Santos
.LINK
    http://www.sqlmusings.com
#>
param([string]$instanceName,[string]$dbName)

$query = @"
TRUNCATE TABLE Test.Person
"@

#check number of records
Invoke-Sqlcmd -Query $query `
-ServerInstance $instanceName `
-Database $dbName
}


function Get-PersonCount {
<#
.SYNOPSIS
    Very simple function to get number
    of records in Test.Person
.NOTES
    Author     : Donabel Santos
.LINK
    http://www.sqlmusings.com
#>
param([string]$instanceName,[string]$dbName)
$query = @"
SELECT COUNT(*) AS NumRecords
FROM Test.Person
"@

#check number of records
Invoke-Sqlcmd -Query $query `
-ServerInstance $instanceName `
-Database $dbName
}
```

3. Add the following script and run it:

```
#let's clean up the Test.Person table first
Truncate-Table $instanceName $dbName

$server = "KERRIGAN"
$table = "AdventureWorks2008R2.Test.Person"
$importfile = "C:\Temp\Exports\AdventureWorks2008R2.Person.Person.
csv"


#command to import from csv
$cmdimport = "bcp $($table) in `"$($importfile)`" -S$server -T -c
-t `"|`" -r `"\n`" "

<#
$cmdimport gives you something like this:
bcp AdventureWorks2008R2.Test.Person in "C:\Temp\Exports\
AdventureWorks2008R2.Person.Person.csv" -SKERRIGAN -T -c -t "|" -r
"\n"
#>

#run the import command
Invoke-Expression $cmdimport

#delay 1 sec, give server some time to import records
#sleep helps us avoid race conditions
Start-Sleep -s 2

Get-PersonCount $instanceName $dbName
```

How it works...

Performing a bulk import using `bcp` is a straightforward task—we need to use the `Invoke-Expression` cmdlet and pass in the `bcp` command. In this recipe, however, we have cleaned up our script a little bit and have started off with a couple of helper functions.

The first helper function, `Truncate-Table`, is a simple helper function that truncates the `Test.Person` table to which we want to import the records. This function passes the `TRUNCATE TABLE` command to SQL Server using the `Invoke-Sqlcmd` cmdlet. To use this function, simply call:

```
Truncate-Table $instanceName $dbName
```

The second helper function, `Get-PersonCount`, simply returns a count of the records that have been imported into the `Test.Person` table. This also uses the `Invoke-Sqlcmd` cmdlet. To invoke the function, use the following code:

```
Get-PersonCount $instanceName $dbName
```

The core of this recipe is with the construction of the `bcp` import command:

```
$server = "KERRIGAN"
$table = "AdventureWorks2008R2.Test.Person"
$importfile = "C:\Temp\Exports\AdventureWorks2008R2.Person.Person.csv"

#command to import from csv
$cmdimport = "bcp " + $table + " in " + '"' + $importfile + '"' + " -S
$server -T -c -t `"|`" -r `"\n`" "
```

This will give us the `bcp` command that points to the import file; it specifies the pipe as the field delimiter and newline as the row delimiter:

```
bcp AdventureWorks2008R2.Test.Person in "C:\Temp\Exports\
AdventureWorks2008R2.Person.Person.csv" -T -c -t "|" -r "\n"
```

Once this command is constructed, we just need to pass it to the `Invoke-Sqlcmd` expression:

```
Invoke-Expression $cmdimport
```

We also added a little bit of delay here using the `Start-Sleep` cmdlet, with a sleep interval of 2 seconds, to allow `INSERT` to happen before we count the records. This is a very simplistic way to avoid race conditions, but for our purposes in this recipe it is sufficient.

See also

- The *Performing bulk import using BULK INSERT* recipe
- The *Performing bulk export using bcp* recipe